

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

Introduction

With the introduction of new devices and extended character sets, new features may be added to this International Standard. Subclauses in the language and library clauses warn implementors and programmers of usages which, though valid in themselves, may conflict with future additions.

Certain features are *obsolescent*, which means that they may be considered for withdrawal in future revisions of this International Standard. They are retained because of their widespread use, but their use in new implementations (for implementation features) or new programs (for language [6.11] or library features [7.26]) is discouraged.

This International Standard is divided into four major subdivisions:

- preliminary elements (clauses 1–4);
- the characteristics of environments that translate and execute C programs (clause 5);
- the language syntax, constraints, and semantics (clause 6);
- the library facilities (clause 7).

Examples are provided to illustrate possible forms of the constructions described. Footnotes are provided to emphasize consequences of the rules described in that subclause or elsewhere in this International Standard. References are used to refer to other related subclauses. Recommendations are provided to give advice or guidance to implementors. Annexes provide additional information and summarize the information contained in this International Standard. A bibliography lists documents that were referred to during the preparation of the standard.

The language clause (clause 6) is derived from “The C Reference Manual”.

The library clause (clause 7) is based on the 1984 */usr/group Standard*.

1. Effort invested in producing the C Standard	6
2. Updates to C90	8
3. Introduction	11
4. Translation environment	13
4.1. Developer expectations	13
4.2. The language specification	14
4.3. Implementation products	14
4.4. Translation technology	15
4.4.1. Translator optimizations	17
5. Execution environment	19
5.1. Host processor characteristics	20
5.1.1. Overcoming performance bottlenecks	23
5.2. Runtime library	26
6. Measuring implementations	26
6.1. SPEC benchmarks	26
6.2. Other benchmarks	27
6.3. Processor measurements	28
7. Introduction	28
8. Source code cost drivers	29
8.1. Guideline cost/benefit	30
8.1.1. What is the cost?	30
8.1.2. What is the benefit?	30
8.1.3. Safer software?	31
8.2. Code development’s place in the universe	31
8.3. Staffing	32
8.3.1. Training new staff	33
8.4. Return on investment	33
8.4.1. Some economics background	34
8.4.1.1. Discounting for time	34
8.4.1.2. Taking risk into account	35
8.4.1.3. Net Present Value	35

8.4.1.4. Estimating discount rate and risk	36
8.5. Reusing software	36
8.6. Using another language	36
8.7. Testability	37
8.8. Software metrics	39
9. Background to these coding guidelines	39
9.1. Culture, knowledge, and behavior	40
9.1.1. Aims and motivation	43
9.2. Selecting guideline recommendations	44
9.2.1. Guideline recommendations must be enforceable	47
9.2.1.1. Uses of adherence to guidelines	48
9.2.1.2. Deviations	48
9.2.2. Code reviews	49
9.3. Relationship among guidelines	50
9.4. How do guideline recommendations work?	50
9.5. Developer differences	51
9.6. What do these guidelines apply to?	52
9.7. When to enforce the guidelines	53
9.8. Other coding guidelines documents	54
9.8.1. Those that stand out from the crowd	55
9.8.1.1. Bell Laboratories and the 5ESS	55
9.8.1.2. MISRA	56
9.8.2. Ada	56
9.9. Software inspections	57
10. Applications	58
10.1. Impact of application domain	58
10.2. Application economics	58
10.3. Software architecture	59
10.3.1. Software evolution	59
11. Developers	60
11.1. What do developers do?	60
11.1.1. Program understanding, not	61
11.1.1.1. Comprehension as relevance	63
11.1.2. The act of writing software	64
11.2. Productivity	64
12. The new(ish) science of people	65
12.1. Brief history of cognitive psychology	65
12.2. Evolutionary psychology	66
12.3. Experimental studies	66
12.3.1. The importance of experiments	67
12.4. The psychology of programming	67
12.4.1. Student subjects	67
12.4.2. Other experimental issues	68
12.5. What question is being answered?	68
12.5.1. Base rate neglect	69
12.5.2. The conjunction fallacy	70
12.5.3. Availability heuristic	72
13. Categorization	73
13.1. Category formation	75
13.1.1. The Defining-attribute theory	76
13.1.2. The Prototype theory	77
13.1.3. The Exemplar-based theory	77
13.1.4. The Explanation-based theory	77
13.2. Measuring similarity	77

- 13.2.1. Predicting categorization performance 79
- 13.3. Cultural background and use of information 82
- 14. Decision making 83
 - 14.1. Decision-making strategies 83
 - 14.1.1. The weighted additive rule 84
 - 14.1.2. The equal weight heuristic 84
 - 14.1.3. The frequency of good and bad features heuristic 84
 - 14.1.4. The majority of confirming dimensions heuristic 85
 - 14.1.5. The satisficing heuristic 85
 - 14.1.6. The lexicographic heuristic 85
 - 14.1.6.1. The elimination-by-aspects heuristic 86
 - 14.1.7. The habitual heuristic 86
 - 14.2. Selecting a strategy 87
 - 14.2.1. Task complexity 87
 - 14.2.2. Response mode 87
 - 14.2.3. Information display 88
 - 14.2.4. Agenda effects 88
 - 14.2.5. Matching and choosing 89
 - 14.3. The developer as decision maker 89
 - 14.3.1. Cognitive effort vs. accuracy 90
 - 14.3.2. Which attributes are considered important? 90
 - 14.3.3. Emotional factors 91
 - 14.3.4. Overconfidence 91
 - 14.4. The impact of guideline recommendations on decision making 93
 - 14.5. Management's impact on developers' decision making 93
 - 14.5.1. Effects of incentives 93
 - 14.5.2. Effects of time pressure 94
 - 14.5.3. Effects of decision importance 94
 - 14.5.4. Effects of training 94
 - 14.5.5. Having to justify decisions 95
 - 14.6. Another theory about decision making 95
- 15. Expertise 96
 - 15.1. Knowledge 97
 - 15.1.1. Declarative knowledge 98
 - 15.1.2. Procedural knowledge 98
 - 15.2. Education 98
 - 15.2.1. Learned skills 99
 - 15.2.2. Cultural skills 99
 - 15.3. Creating experts 99
 - 15.3.1. Transfer of expertise to different domains 100
 - 15.4. Expertise as mental set 100
 - 15.5. Software development expertise 100
 - 15.6. Software developer expertise 101
 - 15.6.1. Is software expertise worth acquiring? 103
 - 15.7. Coding style 103
- 16. Human characteristics 104
 - 16.1. Physical characteristics 107
 - 16.2. Mental characteristics 107
 - 16.2.1. Computational power of the brain 108
 - 16.2.2. Memory 109
 - 16.2.2.1. Visual manipulation 114
 - 16.2.2.2. Longer term memories 114
 - 16.2.2.3. Serial order 116
 - 16.2.2.4. Forgetting 116

16.2.2.5. Organized knowledge	118
16.2.2.6. Memory accuracy	118
16.2.2.7. Errors caused by memory overflow	119
16.2.2.8. Memory and code comprehension	119
16.2.2.9. Memory and aging	120
16.2.3. Attention	120
16.2.4. Automatization	121
16.2.5. Cognitive switch	122
16.2.6. Cognitive effort	123
16.2.7. Human error	124
16.2.7.1. Skill-based mistakes	125
16.2.7.2. Rule-based mistakes	125
16.2.7.3. Knowledge-based mistakes	125
16.2.7.4. Detecting errors	126
16.2.7.5. Error rates	126
16.2.8. Heuristics and biases	126
16.2.8.1. Reasoning	127
16.2.8.2. Rationality	127
16.2.8.3. Risk asymmetry	127
16.2.8.4. Framing effects	129
16.2.8.5. Context effects	129
16.2.8.6. Endowment effect	130
16.2.8.7. Representative heuristic	131
16.2.8.8. Anchoring	133
16.2.8.9. Belief maintenance	133
16.2.8.10. Confirmation bias	138
16.2.8.11. Age-related reasoning ability	140
16.3. Personality	140
17. Introduction	141
17.1. Characteristics of the source code	142
17.2. What source code to measure?	142
17.3. How were the measurements made?	143

Commentary

This book is about the latest version of the C Standard, ISO/IEC 9899:1999 plus TC1, TC2 and TC3 (these contain wording changes derived from WG14's responses to defect reports). It is structured as a detailed, [defect report](#) systematic analysis of that entire language standard (clauses 1–6 in detail; clause 7, the library, is only covered briefly). A few higher-level themes run through all this detail, these are elaborated on below. This book is driven by existing developer practices, not ideal developer practices (whatever they might be). How developers use computer languages is not the only important issue; the writing of translators for them and the characteristics of the hosts on which they have to be executed are also a big influence on the language specification.

Every sentence in the C Standard appears in this book (under the section heading C99). Each of these sentences are followed by a Commentary section, and sections dealing with C90, C++, Other Languages, Common Implementations, Coding Guidelines, Example, and Usage as appropriate. A discussion of each of these sections follows.

Discussions about the C language (indeed all computer languages), by developers, are often strongly influenced by the implementations they happen to use. Other factors include the knowledge, beliefs and biases (commonly known as folklore, or idiom) acquired during whatever formal education or training developers have had and the culture of the group that they current work within. In an attempt to simplify [culture of C](#) discussions your author has attempted to separate out these various threads.

Your author has found that a common complaint made about his discussion of C is that it centers on what

the standard says, not on how particular groups of developers use the language. No apology is made for this outlook. There can be no widespread discussion about C until all the different groups of developers start using consistent terminology, which might as well be that of the standard. While it is true that your author's involvement in the C Standards' process and association with other like-minded people has resulted in a strong interest in unusual cases that rarely, if ever, occur in practice, he promises to try to limit himself to situations that occur in practice, or at least only use the more obscure cases when they help to illuminate the meaning or intent of the C Standard.

No apologies are given for limiting the discussion of language extensions. If you want to learn the details of specific extensions, read your vendor's manuals.

Always remember the definitive definition is what the words in the C Standard say. In responding to defect reports the C committee have at times used the phrase *the intent of the Committee*. This phrase has been used when the wording in the standard is open to more than one possible interpretation and where committee members can recall discussions (via submitted papers, committee minutes, or committee email) in which the intent was expressed. The Committee has generally resisted suggestions to rewrite existing, unambiguous, wording to reflect intent (when the wording has been found to specify different behavior than originally intended).

defect report

Rationale

As well as creating a standards document the C committee also produced a rationale. This rationale document provides background information on the thinking behind decisions made by the Committee.

Wording that appears within a sectioned area like this wording is a direct quote from the rationale (the document used was WG14/N937, dated 17 March 2001).

No standard is perfect (even formally defined languages contain mistakes and ambiguities^[215]). There is a mechanism for clarifying the wording in ISO standards, defect reports (DRs as they are commonly called). The text of C99 DRs are called out where applicable.

defect report

1 Effort invested in producing the C Standard

The ANSI Committee which produced C90, grew from 13 members at the inaugural meeting, in June 1983, to around 200 members just prior to publication of the first Standard. During the early years about 20 people would attend meetings. There was a big increase in numbers once drafts started to be sent out for public review and meeting attendance increased to 50 to 60 people. Meetings occurred four times a year for six years and lasted a week (in the early years meetings did not always last a week). People probably had to put, say, a week's effort into reading papers and preparing their positions before a meeting. So in round numbers let's say:

$$\begin{aligned} &(20 \text{ people} \times 1.3 \text{ weeks} \times 3 \text{ meetings} \times 1 \text{ years}) + \\ &(20 \text{ people} \times 1.7 \text{ weeks} \times 4 \text{ meetings} \times 2 \text{ years}) + \\ &(50 \text{ people} \times 2.0 \text{ weeks} \times 4 \text{ meetings} \times 3 \text{ years}) \Rightarrow 1,540 \text{ person weeks (not quite 30 years)} \end{aligned}$$

What about the 140 people not included in this calculation— how much time did they invest? If they spent just a week a year keeping up with the major issues, then we have 16 person years of effort. On top of this we have the language users and implementors reviewing drafts that were made available for public review. Not all these sent in comments to the Committee, but it is not hard to imagine at least another 4 person years of effort. This gives the conservative figure of 50 person years of effort to produce C90.

Between the publication of C90 and starting work on the revision of C99, the C committee met twice a year for three days; meeting attendance tended to vary between 10 and 20. There was also a significant rise in the use of email during this period. There tended to be less preparation work that needed to be done before meetings— say 2 person years of effort.

The C99 work was done at the ISO level, with the USA providing most of the active committee membership. The Committee met twice a year for five years. Membership numbers were lower, at about 20 per meeting.

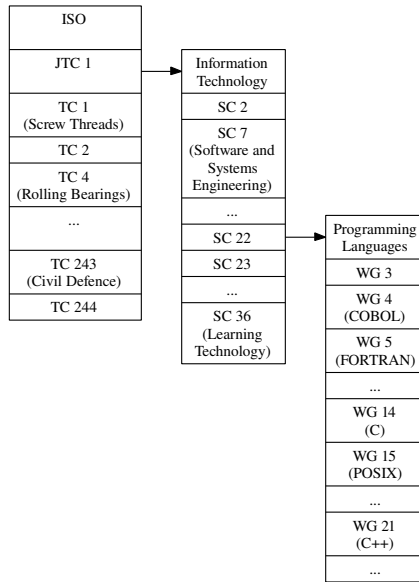


Figure 0.1: The ISO Technical Committee structure— JTC (Joint Technical Committee, with the IEC in this case), TC (Technical Committee), SC (Standards Committee), WG (Working Group).

This gives a figure of 8 person years. During development of C99 there was a significant amount of discussion on the C Standard’s email list; just a week per year equates to more than 2 person years (the UK and Japanese national bodies had active working groups, many of whose members did not attend meetings).

Adding these numbers up gives a conservative total of 62 person years of effort that was invested in the C99 document. This calculation does not include the cost of travelling or any support cost (the document duplication bill for one committee mailing was approximately \$5,000).

The C committee structure

The three letters ISO are said to be derived from the Greek *isos*, meaning “the same” (the official English term used is International Organization for Standardization, not a permutation of these words that gives the ordering ISO). Countries pay to be members of ISO (or to be exact, standards organizations in different countries pay). The size of the payment depends on a country’s gross domestic product (a measure of economic size) and the number of ISO committees they want to actively participate in. Within each country, standards’ bodies (there can be more than one) organize themselves in different ways. In many countries it is possible for their national standards’ body(s) to issue a document as a standard in that country. The initial standards work on C was carried out by one such national body — ANSI (American National Standards Institute). The document they published was only a standard in the USA. This document subsequently went through the process to become an International Standard. As of January 2003, ISO has 138 national standards bodies as members, a turnover of 150 million Swiss Francs, and has published 13,736 International Standards (by 188 technical committees, 550 subcommittees, and 2,937 working groups)(see Figure 0.1).

The documents published by ISO may be formally labeled as having a particular status. These labels include Standard, Technical Report (Type 1, 2, or 3), and a draft of one of these kinds of documents (there are also various levels of draft). The documents most commonly seen by the public are Standards and Type 2 Technical Reports. A Type 2 Technical Report (usually referred to as simply a TR) is a document that is believed to be worth publishing as an ISO Standard, but the material is not yet sufficiently mature to be published as a standard. It is a kind of standard in waiting.

C90

C90 was the first version of the C Standard, known as ISO/IEC 9899:1990(E) (Ritchie^[381] gives a history of prestandard development). It has now been officially superseded by C99. The C90 sections ask the question: What are the differences, if any, between the C90 Standard and the new C99 Standard?

Text such this occurs (with a bar in the margin) when a change of wording can lead to a developer visible change in behavior of a program.

Possible differences include:

- C90 said X was black, C99 says X is white.
- C99 has relaxed a requirement specified in C90.
- C99 has tightened a requirement specified in C90.
- C99 contains a construct that was not supported in C90.

If a construct is new in C99 this fact is only pointed out in the first sentence of any paragraph discussing it. This section is omitted if the wording is identical (word for word, or there are minor word changes that do not change the semantics) to that given in C99. Sometimes sentences have remained the same but have changed their location in the document. Such changes have not been highlighted.

X3J11

The first C Standard was created by the US ANSI Committee X3J11 (since renamed as NCITS J11). This document is sometimes called C89 after its year of publication as an ANSI standard (The shell and utilities portion of POSIX^[183] specifies a c89 command, even although this standard references the ISO C Standard, not the ANSI one.). The published document was known as ANSI X3.159–1989.

This ANSI standard document was submitted, in 1990, to ISO for ratification as an International Standard. Some minor editorial changes needed to be made to the document to accommodate ISO rules (a sed script was used to make the changes to the troff sources from which the camera-ready copy of the ANSI and ISO standards was created). For instance, the word Standard was replaced by International Standard and some major section numbers were changed. More significantly, the Rationale ceased to be included as part of the document (and the list of names of the committee members was removed). After publication of this ISO standard in 1990, ANSI went through its procedures for withdrawing their original document and adopting the ISO Standard. Subsequent purchasers of the ANSI standard see, for instance, the words International Standard not just Standard.

2 Updates to C90

defect report

Part of the responsibility of an ISO Working Group is to provide answers to queries raised against any published standard they are responsible for. During the early 1990s, the appropriate ISO procedure seemed to be the one dealing with defects, and it was decided to create a Defect Report log (entries are commonly known as *DRs*). These procedures were subsequently updated and defect reports were renamed *interpretation requests* by ISO. The C committee continues to use the term *defect* and DR, as well as the new term *interpretation request*.

Standards Committees try to work toward a publication schedule. As the (self-imposed) deadline for publication of the C Standard grew nearer, several issues remained outstanding. Rather than delay the publication date, it was agreed that these issues should be the subject of an Amendment to the Standard. The purpose of this Amendment was to address issues from Denmark (readable trigraphs), Japan (additional support for wide character handling), and the UK (tightening up the specification of some constructs whose wording was considered to be ambiguous). The title of the Amendment was *C Integrity*.

As work on DRs (this is how they continue to be referenced in the official WG14 log) progressed, it became apparent that the issues raised by the UK, to be handled by the Amendment, were best dealt with via these same procedures. It was agreed that the UK work item would be taken out of the Amendment and converted into a series of DRs. The title of the Amendment remained the same even though the material that promoted the choice of title was no longer included within it.

To provide visibility for those cases in which a question had uncovered problems with wording in the published standard the Committee decided to publish collections of DRs. The ISO document containing such corrections is known as a Technical Corrigendum (*TC*) and two were published for C90. A TC is normative and contains edits to the existing standard's wording only, not the original question or any rationale behind the decision reached. An alternative to a TC is a Record of Response (*RR*), a non-normative document.

Wording from the Amendment, the TCs and decisions on defect reports that had not been formally published were integrated into the body of the C99 document.

A determined group of members of X3J11, the ANSI Committee, felt that C could be made more attractive to numerical programmers. To this end it was agreed that this Committee should work toward producing a technical report dealing with numerical issues.

The Numerical C Extensions Group (*NCEG*) was formed on May 10, 1989; its official designation was X3J11.1. The group was disbanded on January 4, 1994. The group produced a number of internal, committee reports, but no officially recognized Technical Reports were produced. Topics covered included: compound literals and designation initializers, extended integers via a header, complex arithmetic, restricted pointers, variable length arrays, data parallel C extensions (a considerable amount of time was spent on discussing the merits of different approaches), and floating-point C extensions. Many of these reports were used as the base documents for constructs introduced into C99.

Support for parallel threads of execution was not addressed by NCEG because there was already an ANSI Committee, X3H5, working toward standardizing a parallelism model and Fortran and C language bindings to it.

C++

Many developers view C++ as a superset of C and expect to be able to migrate C code to C++. While this book does not get involved in discussing the major redesigns that are likely to be needed to make effective use of C++, it does do its best to dispel the myth of C being a subset of C++. There may be a language that is common to both, but these sections tend to concentrate on the issues that need to be considered when translating C source using a C++ translator.

What does the C++ Standard, ISO/IEC 14882:1998(E), have to say about constructs that are in C99?

- *Wording is identical.* Say no more.
- *Wording is similar.* Slight English grammar differences, use of terminology differences and other minor issues. These are sometimes pointed out.
- *Wording is different but has the same meaning.* The sequence of words is too different to claim they are the same. But the meaning appears to be the same. These are not pointed out unless they highlight a C++ view of the world that is different from C.
- *Wording is different and has a different meaning.* Here the C++ wording is quoted, along with a discussion of the differences.
- *No C++ sentence can be associated with a C99 sentence.* This often occurs because of a construct that does not appear in the C++ Standard and this has been pointed out in a previous sentence occurring before this derived sentence.

There is a stylized form used to comment source code associated with C— `/* behavior */`— and C++— `// behavior`.

The precursor to C++ was known as C with Classes. While it was being developed C++ existed in an environment where there was extensive C expertise and C source code. Attempts by Stroustrup to introduce incompatibilities were met by complaints from his users.^[435]

The intertwining of C and C++, in developers mind-sets, in vendors shipping a single translator with a language selection option, and in the coexistence of translation units written in either language making up one program means that it is necessary to describe any differences between the two.

The April 1989 meeting of WG14 was asked two questions by ISO: (1) should the C++ language be standardized, and (2) was WG14 the Committee that should do the work? The decision on (1) was very close, some arguing that C++ had not yet matured sufficiently to warrant being standardized, others arguing that working toward a standard would stabilize the language (constant changes to its specification and implementation were causing headaches for developers using it for mission-critical applications). Having agreed that there should be a C++ Standard WG14 was almost unanimous in stating that they were not the Committee that should create the standard. During April 1991 WG21, the ISO C++ Standard's Committee was formed; they met for the first time two months later.

In places additional background information on C++ is provided. Particularly where different concepts, or terminology, are used to describe what is essentially the same behavior.

In a few places constructs available in C++, but not C, are described. The rationale for this is that a C developer, only having a C++ translator to work with, might accidentally use a C++ construct. Many C++ translators offer a C compatibility mode, which often does little more than switch off support for a few C++ constructs. This description may also provide some background about why things are different in C++.

Everybody has a view point, even the creator of C++, Bjarne Stroustrup. But the final say belongs to the standards' body that oversees the development of language standards, SC22. The following was the initial position.

*Resolutions Prepared at the Plenary Meeting of
ISO/IEC JTC 1/SC22
Vienna, Austria
September 23–29, 1991*

Resolution AK Differences between C and C++

Notwithstanding that C and C++ are separate languages, ISO/IEC JTC1/SC22 directs WG21 to document differences in accordance with ISO/IEC TR 10176.

Resolution AL WG14 (C) and WG21 (C++) Coordination

While recognizing the need to preserve the respective and different goals of C and C++, ISO/IEC JTC1/SC22 directs WG14 and WG21 to ensure, in current and future development of their respective languages, that differences between C and C++ are kept to the minimum. The word "differences" is taken to refer to strictly conforming programs of C which either are invalid programs in C++ or have different semantics in C++.

This position was updated after work on the first C++ Standard had been completed, but too late to have any major impact on the revision of the C Standard.

*Resolutions Prepared at the Eleventh Plenary Meeting of
ISO/IEC JTC 1/SC22
Snekkersten, Denmark
August 24–27, 1998*

Resolution 98-6: Relationship Between the Work of WG21 and that of WG14

Recognizing that the user communities of the C and C++ languages are becoming increasingly divergent, ISO/IEC JTC 1/SC22 authorizes WG21 to carry out future revisions of ISO/IEC 14882:1998 (Programming Language C++) without necessarily adopting new C language features contained in the current revision to ISO/IEC 9899:1990 (Programming Language C) or any future revisions thereof.

ISO/IEC JTC 1/SC22 encourages WG14 and WG21 to continue their close cooperation in the future.

Other Languages

Why are other languages discussed in this book? Developers are unlikely to spend their entire working life using a single language (perhaps some Cobol and Fortran programmers may soon achieve this).

C is not the only programming language in the world (although some developers act as if it were). Characteristics of other languages can help sharpen a developer's comprehension of the spirit (design, flavor, world-view) of C. Some of C's constructs could have been selected in several alternative ways, others interrelate to each other.

The functionality available in C can affect the way an algorithm is coded (not forgetting individual personal differences^[362,363]). Sections of source may only be written that way because that is how things are done in C; they may be written differently, and have different execution time characteristics,^[364] in other languages. Appreciating the effects of C language features in the source they write can be very difficult for developers to do; rather like a fish trying to understand the difference between water and dry land.

Some constructs are almost universal to all programming languages, others are unique to C (and often C++). Some constructs are common to a particular class of languages— algorithmic, functional, imperative, formal, and so on. The way things are done in C is not always the only way of achieving the same result, or the same algorithmic effect. Sometimes C is unique. Sometimes C is similar to what other languages do. Sometimes there are languages that do things very differently from C, either in implementing the same idea, or in having a different view of the world.

It is not the intent to claim that C or any other language is better or worse because it has a particular design philosophy, or contains a particular construct. Neither is this subsection intended as a survey of what other languages do. No attempt is made to discuss any other language in any way apart from how it is similar or different from C. Other languages are looked at from the C point of view.

Developers moving from C to another language will, for a year or so (or longer depending on the time spent using the new language), tend to use that language in a C-like style (much the same as people learning English tend to initially use the grammar and pronunciations of their native language; something that fluent speakers have no trouble hearing).

Your author's experience with many C developers is that they tend to have a *C is the only language worth knowing attitude*. This section is unlikely to change that view and does not seek to. Some knowledge of how other languages do things never hurt.

There are a few languages that have stood the test of time, Cobol and Fortran for example. While Pascal and Ada may have had a strong influence on the thinking about how to write maintainable, robust code, they have come and gone in a relatively short period of time. At the time of this writing there are six implementations of Ada 95. A 1995 survey^[173] of language usage found 49.5 million lines of Ada 83 (C89 32.5 million, other languages 66.18 million) in DoD weapon systems. The lack of interest in the Pascal standard is causing people to ask whether it should be withdrawn as a recognized standard (ISO rules require that a standard be reviewed every five years). The Java language is making inroads into the embedded systems market (the promise of it becoming the lingua franca of the Internet does not seem to have occurred). It is also trendy, which keeps it in the public eye. Lisp continues to have a dedicated user base 40 years after its creation. A paper praising its use, over C, has even been written.^[121]

The references for the other languages mentioned in this book are: Ada,^[189] Algol 68,^[473] APL,^[194] BCPL,^[378] CHILL,^[196] Cobol,^[181] Fortran,^[187] Lisp^[192] (Scheme^[217]), Modula-2,^[190] Pascal,^[186] Perl,^[485] PL/1,^[180] Snobol 4,^[152] and SQL.^[188]

References for the implementation of languages that have significant differences from C include APL,^[55] functional languages,^[347] and ML.^[18]

Common Implementations

3 Introduction

This subsection gives an overview of translator implementation issues. The specific details are discussed in the relevant sentence. The following are the main issues.

- *Translation environment*. This environment is defined very broadly here. It not only includes the language specification (dialects and common extensions), but customer expectations, known translation

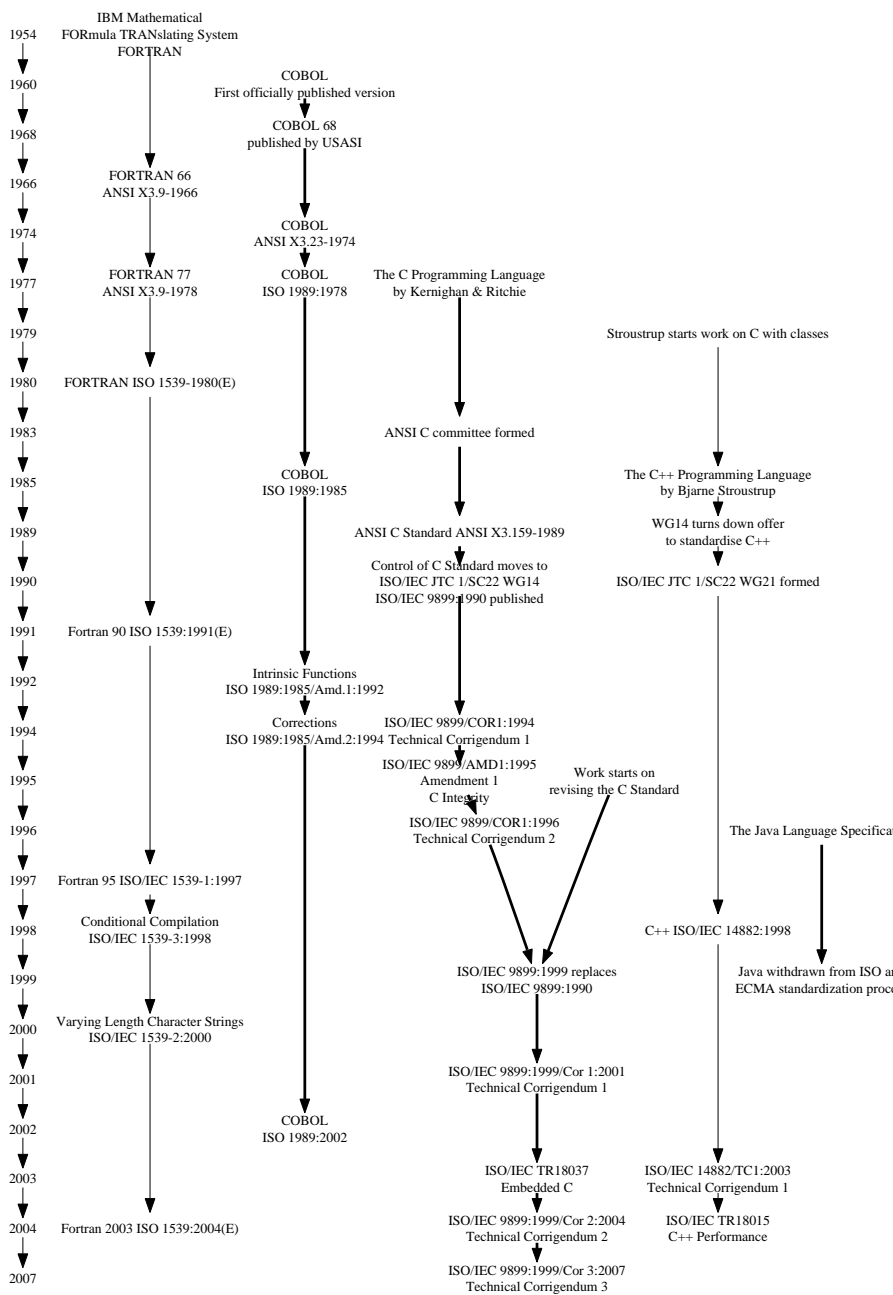


Figure 0.2: Outline history of the C language and a few long-lived languages. (Backus^[21] describes the earliest history of Fortran.)

technology and the resources available to develop and maintain translators. Like any other application development project, translators have to be written to a budget and time scale.

- *Execution environment.* This includes the characteristics of the processor that will execute the program image (instruction set, number of registers, memory access characteristics, etc.), and the runtime interface to the host environment (storage allocation, function calling conventions, etc.).
- *Measuring implementations.* Measurements on the internal working of translators is not usually published. However, the execution time characteristics of programs, using particular implementations, is of great interest to developers and extensive measurements are made (many of which have been published).

4 Translation environment

The translation environment is where developers consider their interaction with an implementation to occur. Any requirement that has existed for a long period of time (translators, for a variety of languages, have existed for more than 40 years; C for 25 years) establishes practices for how things should be done, accumulates a set of customer expectations, and offers potential commercial opportunities.

Although the characteristics of the language that need to be translated have not changed significantly, several other important factors have changed. The resources available to a translator have significantly increased and the characteristics of the target processors continue to change. This increase in resources and need to handle new processor characteristics has created an active code optimization research community.

4.1 Developer expectations

Developers have expectations about what language constructs mean and how implementations will process them. At the very least developers expect a translator to accept their existing source code and generate to a program image from it, the execution time behavior being effectively the same as the last implementation they used. Implementation vendors want to meet developer expectations whenever possible; it reduces the support overhead and makes for happier customers. Authors of translators spend a lot of time discussing what their customers expect of their product; however, detailed surveys of customer requirements are rarely carried out. What is available is existing source code. It is this existing code base that is often taken as representing developers expectations (translators should handle it without complaint, creating programs that deliver the expected behavior).

developer
expectations

Three commonly encountered expectations are good performance, low code expansion ratio, and no surprising behavior; the following describes these expectations in more detail.

1. *C has a reputation for efficiency.* It is possible to write programs that come close to making optimum usage of processor resources. Writing such code manually relies on knowledge of the processor and how the translator used maps constructs to machine code. Very few developers know enough about these subjects to be able to consistently write very efficient programs. Your author sometimes has trouble predicting the machine code that would be generated when using the compilers he had written. As a general rule, your author finds it safe to say that any ideas developers have about the most efficient construct to use, at the statement level, are wrong. A cost effective solution is to not worry about statement level efficiency issues and let the translator look after things.
2. *C has a reputation for compactness.* The ratio of machine code instructions per C statement is often a small number compared to other languages. It could be said that C is a WYSIWYG language, the mapping from C statement to machine code being simple and obvious (leaving aside what an optimizer might subsequently do). This expectation was used by some members of WG14 as an argument against allowing the equality operator to have operands with structure type; a single operator potentially causing a large amount of code, a comparison for each member, to be generated. The introduction of the **inline** function-specifier has undermined this expectation to some degree (depending on whether **inline** is thought of as a replacement for function-like macros, or the inlining of functions that would not have been implemented as macros).

function
specifier
syntax
macro
function-like

3. *C* has a reputation for being a consistent language. Developers can usually predict the behavior of the code they write. There are few dark corners whose accidental usage can cause constructs to behave in unexpected ways. While the C committee can never guarantee that there would never be any surprising behaviors, it did invest effort in trying to ensure that the least-surprising behaviors occurred.

4.2 The language specification

The C Standard does not specify everything that an implementation of it has to do. Neither does it prevent vendors from adding their own extensions. C is not a registered trademark that is policed to ensure implementations follow its requirements; unlike Ada, which until recently was a registered trademark, owned by the US Department of Defense, which required that an implementation pass a formal validation procedure before allowing it to be called Ada. The C language also has a history— it existed for 13 years before a formally recognized standard was ratified.

The commercial environments in which C was originally used have had some influence on its specification. The C language started life on comparatively small platforms and the source code of a translator (pcc, the portable C compiler^[201]) was available for less than the cost of writing a new one. Smaller hardware vendors without an established customer base, were keen to promote portability of applications to their platform. Thus, there were very few widely accepted extensions to the base language. In this environment vendors tended to compete more in the area of available library functions. For this reason, significant developer communities, using different dialects of C, were not created. Established hardware vendors are not averse to adding language extensions specific to their platforms, which resulted in several widely used dialects of both Cobol and Fortran.

Implementation vendors have found that they can provide a product that simply follows the requirements contained in the C Standard. While some vendors have supplied options to support for some prestandard language features, the number of these features is small.

Although old source code is rarely rewritten, it still needs a host to run on. The replacement of old hosts by newer ones means that either existing source has to be ported, or new software acquired. In both cases it is likely that the use of prestandard C constructs will diminish. Many of the programs making use of C language dialects, so common in the 1980s, are now usually only seen executing on very old hosts. The few exceptions are discussed in the relevant sentences.

4.3 Implementation products

Translators are software products that have customers like any other application. The companies that produce them have shareholders to satisfy and, if they are to stay in business, need to take commercial issues into account. It has always been difficult to make money selling translators and the continuing improvement in the quality of Open Source C translators makes it even harder. Vendors who are still making most of their income by selling translators, as opposed to those who have to supply one as part of a larger sale, need to be very focused and tend to operate within specific markets.^[498] For instance, some choose to concentrate on the development process (speed of translation, integrated development environment, and sophisticated debugging tools), others on the performance of the generated machine code (Kuck & Associates, purchased by Intel, for parallelizing scientific and engineering applications, Code Play for games developers targeting the Intel x86 processor family). There are even specialists within niches. For instance, within the embedded systems market Byte Craft concentrates on translators for 8-bit processors. Vendors who are still making most of their income from selling other products (e.g., hardware or operating systems) sometimes include a translator as a loss leader. Given its size there is relatively little profit for Microsoft in selling a C/C++ translator; having a translator gives the company greater control over its significantly more profitable products (written in those languages) and, more importantly, mind-share of developers producing products for its operating systems.

It is possible to purchase a license for a C translator front-end from several companies. While writing one from scratch is not a significant undertaking (a few person years), writing anything other than a straightforward code generator can require a large investment. By their very nature, many optimization techniques deal with special cases, looking to fine-tune the use of processor resources. Ensuring that correct code is

generated, for all the myriad different combinations of events that can occur, is very time-consuming and expensive.

The performance of generated machine code is rarely the primary factor in developer selection of which translator to purchase, if more than one is available to choose from. Factors such as implicit Vendor preference (it is said that nobody is sacked for buying Microsoft), preference for the development environment provided, possessing existing code that is known to work well with a particular vendor's product, and many other possible issues. For this reason optimization techniques often take many years to find their way from published papers to commercial products, if at all.^[382]

Companies whose primary business is the sale of translators do not seem to grow beyond a certain point. The largest tend to have a turnover in the tens of millions of dollars. The importance of translators to companies in other lines of business has often led to these companies acquiring translator vendors, both for the expertise of their staff and for their products. Several database companies have acquired translator vendors to use their expertise and technology in improving the performance of the database products (the translators subsequently being dropped as stand-alone products).

Overall application performance is often an issue in the workstation market. Here vendors, such as HP, SGI, and IBM, have found it worthwhile investing in translator technology that improves the quality of generated code for their processors. Potential customers evaluating platforms using benchmarks will be looking at numbers that are affected by both processor and translator performance—the money to be made from multiple hardware sales being significantly greater than that from licensing a translator to relatively few developers. These companies consider it worthwhile to have an in-house translator development group.

GCC, the GNU C compiler^[426] (now renamed the GNU Compiler Collection; the term *gcc* will be used here to refer to the C compiler), was distributed in source code form long before Linux and the rise of the Open Source movement. Its development has been checkered, but it continues to grow from strength to strength. This translator was designed to be easily retargeted to a variety of different processors. Several processor vendors have provided, or funded ports of the back end to their products. Over time the optimizations performed by GCC have grown more sophisticated. This has a lot to do with researchers using GCC as the translator on which to implement and test their optimization ideas. On those platforms where its generated machine code does not rank first in performance, it usually ranks second.

GCC

The source code to several other C translators has also been released under some form of public use license. These include: *lcc*^[132] along with *vpo* (very portable optimizer^[40]), the *SGIPRO C compiler*^[409] (which performs many significant optimizations), the *TenDRA C/C++ project*,^[15] *Watcom*,^[488] *Extensible Interactive C* (an interpreter),^[53] and the *Trimaran compiler system*.^[17]

The lesson to be drawn from these commercial realities is that developers should not expect a highly competitive market in language translators.^[498] Investing large amounts of money in translator development is unlikely to be recouped purely from sales of translators (some vendors make the investment to boost the sales of their processors). Developers need to work with what they are given.

4.4 Translation technology

Translators for C exist within a community of researchers (interested in translation techniques) and also translators for other languages. Some techniques have become generally accepted as the way some construct is best implemented; some are dictated by trends that come and go. This book does not aim to document every implementation technique, but it may discuss the following.

translation
technology

- How implementations commonly map constructs for execution by processors.
- Unusual processor characteristics, which affect implementations.
- Common extensions in this area.
- Possible trade-offs involved in implementing a construct.
- The impact of common processor architectures on the C language.

In the early days of translation technology vendors had to invest a lot of effort simply to get them to run within the memory constraints of the available development environments. Many existed as a collection of separate programs, each writing output to be read by the succeeding phase, the last phase being assembler code that needed to be processed by an assembler.

Ever since the first Fortran translator^[21] the quality of machine code produced has been compared to handwritten assembler. Initially translators were only asked to not produce code that was significantly worse than handwritten assembler; the advantages of not having to retrain developers (in new assembly languages) and rewrite applications outweigh the penalties of less performance. The fact that processors changed frequently, but software did not, was a constant reminder of the advantages of using a machine-independent language. Whether most developers stopped making the comparison against handwritten assembler because fewer of them knew any assembler, or because translators simply got better is an open issue. In some application domains the quality of code produced by translators is nowhere near that of handwritten assembler^[419] and many developers still need to write in machine code to be able to create usable applications.

Much of the early work on translators was primarily concerned with different language constructs and parsing them. A lot of research was done on various techniques for parsing grammars and tools for compressing their associated data tables. The work done at Carnegie Mellon on the PQCC project^[261] introduced many of the ideas commonly used today. By the time C came along there were some generally accepted principles about how a translator should be structured.

footnote

A C translator usually operates in several phases. The first phase (called the *front-end* by compiler writers and often the parser by developers) performs syntax and semantic analysis of the source code and builds a tree representation (usually based on the abstract syntax); it may also map operations to an intermediate form (some translators have multiple intermediate forms, which get progressively lower as constructs proceed through the translation process) that has a lower-level representation than the source code but a higher-level than machine code. The last phase (often called the *back-end* by compiler writers or the *code generator* by developers) takes what is often a high-level abstract machine code (an intermediate code) and maps it to machine code (it may generate assembler or go directly to object code). Operations, such as storage layout and optimizations on the intermediate code, could be part of one of these phases, or be a separate phase (sometimes called the *middle-end* by compiler writers).

storage layout

The advantage of generating machine code from intermediate code is a reduction in the cost of retargeting the translator to a new processor; the front-end remains virtually the same and it is often possible to reuse substantial parts of later passes. It becomes cost effective for a vendor to offer a translator that can generate machine code for different processors from the same source code. Many translators have a single intermediate code. GCC currently has one, called RTL (register transfer language), but may soon have more (a high-level, machine-independent, RTL, which is then mapped to a more machine specific form of RTL). Automatically deriving code generators from processor descriptions^[64] sounds very attractive. However, until recently new processors were not introduced sufficiently often to make it cost effective to remove the human compiler written from the process. The cost of creating new processors, with special purpose instruction sets, is being reduced to the point where custom processors are likely to become very common and automatic derivation of code generators is essential to keep these costs down.^[246,259]

The other advantage of breaking the translator into several components is that it offers a solution to the problem caused by a common host limitation. Many early processors limited the amount of memory available to a program (64 K was a common restriction). Splitting a translator into independent components (the preprocessor was usually split off from the syntax and semantics processing as a separate program) enabled each of them to occupy this limited memory in turn. Today most translators have many megabytes of storage available to them; however, many continue to have internal structures designed when storage limitations were an important issue.

There are often many different ways of translating C source into machine code. Developers invariably want their programs to execute as quickly as possible and have been sold on the idea of translators that

perform code optimization. There is no commonly agreed on specification for exactly what a translator needs to do to be classified as optimizing, although claims made in a suitably glossy brochure is often sufficient for many developers.

4.4.1 Translator optimizations

Traditionally optimizations have been aimed at reducing the time needed to execute a program (this is what the term *increasing program performance* is usually intended to mean) or reducing the size of the program image (this usually means the amount of storage occupied during program execution—consisting of machine code instructions, some literal values, and object storage). Many optimizations have the effect of increasing performance and reducing size. However, there are some optimizations that involve making a trade-off between performance and size.

The growth in mobile phones and other hand-held devices containing some form of processor have created a new optimization requirement—power minimization. Software developers want to minimize the amount of electrical power required to execute a program. This optimization requirement is likely to be new to readers; for this reason a little more detail is given at the end of this subsection.

Some of the issues associated with generating optimal machine code for various constructs are discussed within the sentences for those constructs. In some cases transformations are performed on a relatively high-level representation and are relatively processor-independent (see Bacon, Graham, and Sharp^[22] for a review). Once the high-level representation is mapped to something closer to machine code, the optimizations can become very dependent on the characteristics of the target processor (Bonk and Rude^[48] look at number crunchers). The general techniques used to perform optimizations at different levels of representation can be found in various books.^[4, 132, 154, 312]

The problems associated with simply getting a translator written became tractable during the 1970s. Since then the issues associated with translators have been the engineering problem of being able to process existing source code and the technical problem of generating high-quality machine code. The focus of code optimization research continues to evolve. It started out concentrating on expressions, then basic blocks, then complete functions and now complete programs. Hardware characteristics have not stood still either. Generating optimized machine code can now require knowledge of code and data cache behaviors, speculative execution, dependencies between instructions and their operands. There is also the issue of processor vendors introducing a range of products, all supporting the same instruction set but at different price levels and different internal performance enhancements; optimal instruction selection can now vary significantly across a single processor family.

Sometimes all the information about some of the components used by a program will not be known until it is installed on the particular host that executes it; for instance, any additional instructions supported over those provided in the base instruction set for that processor, the relative timings of instructions for that processor model, and the version of any dynamic linked libraries. These can also change because of other systems software updates. Also spending a lot of time during application installation generating an optimal executable program is not always acceptable to end users. One solution is to perform optimizations on the program while it is executing. Because most of the execution time usually occurs within a small percentage of a program's machine code, an optimizer only needs to concentrate on these areas. Experimental systems are starting to deliver interesting results.^[221]

Thorup^[454] has shown that a linear (in the number of nodes and vertices in the control flow graph) algorithm for register allocation exists that is within a factor of seven (six if no short-circuit evaluation is used) of the optimal solution for any C program that does not contain `gotos`.

One way of finding optimal instruction sequences is to generate all possible sequences and to select the optimal one that provides the desired input to output transformation. Massalin^[279] built a *superoptimizer* to do just that; it worked off-line and was not intended to be used to generate instruction sequences during translation. Bansal and Aiken^[32] built a superoptimizer that is intended to be used within a translator to find optimal instruction sequences. The tools used various strategies to reduce the search space, e.g., pruning instruction sequences known to be nonoptimal and maintaining a database of previously generated optimal

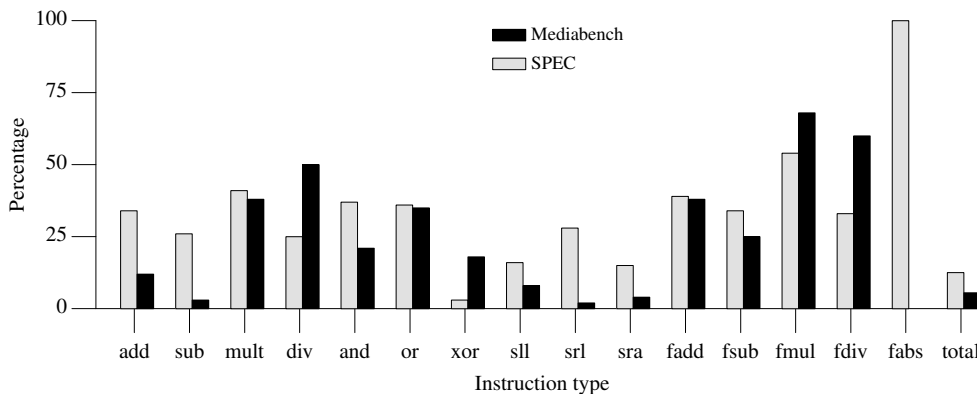


Figure 0.3: Dynamic frequency, percentage calculated over shown instructions (last column gives percentage of these instruction relative to all instructions executed) during execution of the SPEC and MediaBench benchmarks of some computational oriented instructions. Adapted from Yi and Lilja.^[501]

sequences.

Code optimization is a, translation time, resource-hungry process. To reduce the quantity of analysis that needs to be performed, optimizers have started to use information on a program's runtime characteristics. This profile information enables optimizers to concentrate resources on frequently executed sections of code (it also provides information on the most frequent control flow path in conditional statements, enabling the surrounding code to be tuned to this most likely case).^[155,500] However, the use of profile information does not always guarantee better performance.^[244]

The stability of execution profiles, that is the likelihood that a particular data set will always highlight the same sections of a program as being frequently executed is an important issue. A study by Chilimbi^[70] found that data reference profiles, important for storage optimization, were stable, while some other researchers have found that programs exhibit different behaviors during different parts of their execution.^[402]

Optimizers are not always able to detect all possible savings. A study by Yi and Lilja^[501] traced the values of instruction operands during program execution. They found that a significant number of operations could have been optimized (see Figure 0.3) had one of their operand values been known at translation time (e.g., adding/subtracting zero, multiplying by 1, subtracting/dividing two equal values, or dividing by a power of 2).

Power consumption

The following discussion is based one that can be found in Hsu, Kremer and Hsiao.^[178] The dominant source of power consumption in digital CMOS circuits (the fabrication technology used in mass-produced processors) is the dynamic power dissipation, P , which is based on three factors:

$$P \propto CV^2F \quad (0.1)$$

where C is the effective switching capacitance, V the supply voltage, and F the clock speed. A number of technical issues prevent the voltage from being arbitrarily reduced, but there are no restrictions on reducing the clock speed (although some chips have problems running at too low a rate).

For cpu bound programs simply reducing the clock speed does not usually lead to any significant saving in total power consumption. A reduction in clock speed often leads to a decrease in performance and the program takes longer to execute. The product of dynamic power consumption and time taken to execute remains almost unchanged (because of the linear relationship between dynamic power consumption and clock speed). However, random access memory is clocked at a rate that can be an order of magnitude less than the processor clock rate.

For memory-intensive applications a processor can be spending most of its time doing nothing but waiting for the results of load instructions to appear in registers. In these cases a reduction in processor clock rate will have little impact on the performance of a program. Program execution time, T , can be written as:

$$T = T_{cpu_busy} + T_{memory_busy} + T_{cpu_and_mem_busy} \quad (0.2)$$

An analysis (using a processor simulation) of the characteristics of the following code:

```

1  for (j = 0; j < n; j++)
2      for (i = 0; i < n; i++)
3          accu += A[i][j];

```

found that (without any optimization) the percentage of time spent in the various subsystems was: $cpu_busy=0.01\%$, $memory_busy=93.99\%$, $cpu_and_mem_busy=6.00\%$.

Given these performance characteristics, a factor of 10 reduction in the clock rate and a voltage reduction from 1.65 to 0.90 would reduce power consumption by a factor of 3, while only slowing the program down by 1% (these values are based on the Crusoe TM5400 processor).

Performing optimizations changes the memory access characteristics of the loop, as well as potentially reducing the amount of time a program takes to execute. Some optimizations and their effect on the performance of the preceding code fragment include the following:

- Reversing the order of the loop control variables (arrays in C are stored in row-major order) creates spatial locality, and values are more likely to have been preloaded into the cache: $cpu_busy=18.93\%$, $memory_busy=73.66\%$, $cpu_and_mem_busy=7.41\%$ loop control variable array row-major storage order
- Loop unrolling increases the amount of work done per loop iteration (decreasing loop housekeeping overhead and potentially increasing the number of instructions in a basic block): $cpu_busy=0.67\%$, $memory_busy=65.60\%$, $cpu_and_mem_busy=33.73\%$ loop unrolling basic block
- Prefetching data can also be a worthwhile optimization:^[474] $cpu_busy=0.67\%$, $memory_busy=74.04\%$, $cpu_and_mem_busy=25.29\%$

These ideas are still at the research stage^[177] and have yet to appear in commercially available translators (support, in the form of an instruction to change frequency/voltage, also needs to be provided by processor vendors).

At the lowest level processors are built from transistors, which are grouped together to form logic gates. In CMOS circuits power is dissipated in a gate when its output changes (i.e., it goes from 0 to 1, or from 1 to 0). Vendors interested in low power consumption try to minimize the number of gate transitions made during the operation of a processor. Translators can also help here. Machine code instructions consist of sequences of zeros and ones. Differences in bit patterns between adjacent instructions, encountered during program execution, cause gate transitions. The Hamming distance between two binary values (instructions) is the number of places at which their bit settings differ. Ordering instructions to minimize the total Hamming distance over the entire sequence will reduce power consumption in the instruction decoding area of a processor. Simulations based on such a reordering have shown savings of 13% to 20%.^[251]

5 Execution environment

Two kinds of execution environment are specified in the C Standard, hosted and freestanding. These tend to affect implementations in terms of the quantity of resources provided (functionality to support library requirements— e.g., I/O, memory capacity, etc.). environment execution

There are classes of applications that tend to occur in only one of these environments, which can make it difficult to classify an issue as being application- or environment-based.

For hosted environments C programs may need to coexist with programs written in a variety of languages. Vendors often define a set of conventions that programs need to follow; for instance, how parameters are passed. The popularity of C for systems development means that such conventions are often expressed in C terms and the implementations of other languages have to adapt to the C view of how things work.

Existing environments have affected the requirements in the C Standard library. Unlike some languages the C language has tried to take the likely availability of functionality in different environments into account. For instance, the inability of some hosts to support signals has meant that there is no requirement that any signal handling (other than function stubs) be provided by an implementation. Minimizing the dependency on constructs being supported by a host environment enables C to be implemented on a wide variety of platforms. This wide implementability comes at the cost of some variability in supported constructs.

5.1 Host processor characteristics

It is often recommended that developers ignore the details of host processor characteristics. However, the C language was, and continues to be, designed for efficient mapping to commonly available processors. Many of the benchmarks by which processor performance is measured are written in C. A detailed analysis of C needs to include a discussion of processor characteristics.

Many developers continue to show a strong interest in having their programs execute as quickly as possible, and write code that they think will achieve this goal. Developer interest in processor characteristics is often driven by this interest in performance and efficiency. Developer interest in performance could be considered to be part of the culture of programming. It does not seem to be C specific, although this language's reputation for efficiency seems to exacerbate it. There is sometimes a customer-driven requirement for programs to execute within resource constraints (execution time and memory being the most common constrained resources). In these cases detailed knowledge of processor characteristics may help developers tune an application (although algorithmic tuning invariably yields higher returns on investment). However, the information given in this book is at the level of a general overview. Developers will need to read processor vendor's manuals, very carefully, before they can hope to take advantage of processor-specific characteristics by changing how they write source code.

The following are the investment issues, from the software development point of view, associated with processor characteristics:

- Making effective use of processor characteristics usually requires a great deal of effort (for an in-depth tutorial on getting the best out of a particular processor see,^[451] for an example of performance forecasting aimed at future processors see Armstrong and Eigenmann^[20]). The return on investment of this effort is often small (if not zero). Experience shows that few developers invest the time needed to systematically learn about individual processor characteristics. Preferring, instead, to rely on what they already know, articles in magazines, and discussions with other developers. A small amount of misguided investment is no more cost effective than overly excessive knowledgeable investment.
- Processors change more frequently than existing code. Although there are some application domains where it appears that the processor architecture is relatively fixed (e.g., the Intel x86 and IBM 360/370/3080/3090/etc.), the performance characteristics of different members of the same family can still vary dramatically. Within the other domains new processor architectures are still being regularly introduced. The likelihood of a change of processor remains an important issue.
- The commercial availability of translators capable of producing machine code, the performance of which is comparable to that of handwritten assembler (this is not true in some domains;^[419] one study^[469] found that in many cases translator generated machine code was a factor of 5–8 times slower than hand crafted assembler) means that any additional return on developer resource investment is likely to be low.

Commercial and application considerations have caused hardware vendors to produce processors aimed at several different markets. It can be said that there are often family characteristics of processors within a

host processors
introduction

SPEC⁰
benchmarks

translator per-
formance
vs. assembler

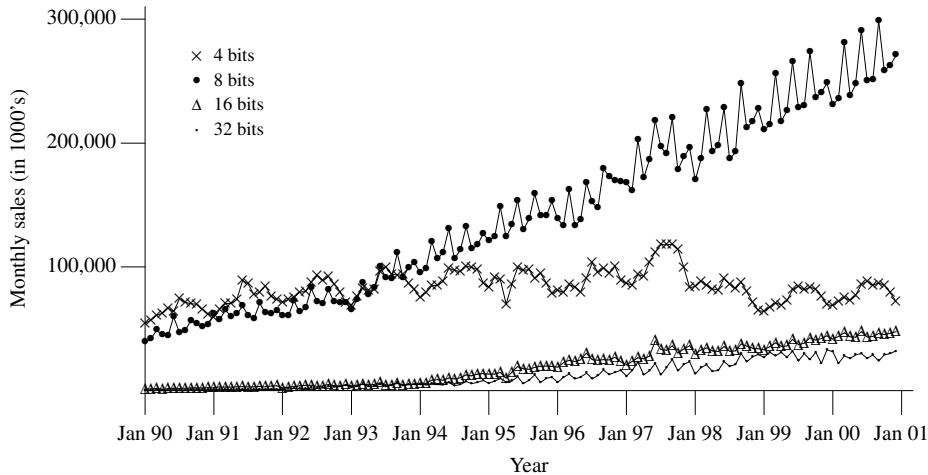


Figure 0.4: Monthly unit sales of microprocessors having a given bus width. Adapted from Turley^[459] (using data supplied by Turley).

given market, although the boundaries are blurred at times. It is not just the applications that are executed on certain kinds of processors. Often translator vendors target their products at specific kinds of processors. For instance, a translator vendor may establish itself within the embedded systems market. The processor architectures can have a dramatic effect on the kinds of problems that machine code generators and optimizers need to concern themselves with. Sometimes the relative performance of programs written in C, compared to handwritten assembler, can be low enough to question the use of C at all.

- *General purpose processors.* These are intended to be capable of running a wide range of applications. The processor is a significant, but not dominant, cost in the complete computing platform. The growing importance of multimedia applications has led many vendors to extend existing architectures to include instructions that would have previously only been found in DSP processors.^[419] The market size can vary from tens of millions (e.g., Intel x86^[445]) to hundreds of millions (e.g., ARM^[445]).
- *Embedded processors.* These are used in situations where the cost of the processor and its supporting chip set needs to be minimized. Processor costs can be reduced by reducing chip pin-out (which reduces the width of the data bus) and by reducing the number of transistors used to build the processor. The consequences of these cost savings are that instructions are often implemented using slower techniques and there may not be any performance enhancers such as branch prediction or caches (or even multiple and divide instructions, which have to be emulated in software). Some vendors offer a range of different processors, others a range of options within a single family, using the same instruction set (i.e., the price of an Intel i960 can vary by an order of magnitude, along with significant differentiation in its performance, packaging, and level of integration). The total market size is measured in billions of processors per year (see Figure 0.4).
- *Digital Signal Processors (DSP).* As the name suggests, these processors are designed for manipulating digital signals—for instance, decoding MPEG data streams, sending/receiving data via phone lines, and digital filtering types of applications. These processors are specialized to perform this particular kind of application very well; it is not intended that nondigital signal-processing applications ever execute on them. Traditionally DSPs have been used in applications where dataflow is the dominating factor;^[44] making the provision of handcrafted library routines crucial. Recently new markets, such as telecoms and the automobile industry have started to use DSPs in a big way, and their applications have tended to be dominated by control flow, reducing the importance of libraries. Araújo^[96] contains an up-to-date discussion on generating machine code for DSPs. The total worldwide market in 1999 was 0.6 billion processors;^[445] individual vendors expect to sell hundreds of millions of units.

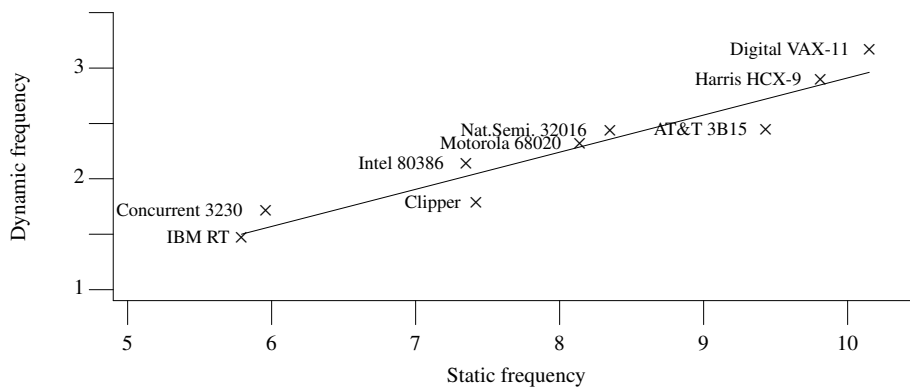


Figure 0.5: Dynamic/static frequency of *call* instructions. Adapted from Davidson.^[94]

- *Application Specific Instruction-set Processors (ASIP)*. Note that the acronym ASIC is often heard, this refers to an Application Specific Integrated Circuit—a chip that may or may not contain an instruction-set processor. These processors are designed to execute a specific program. The general architecture of the processor is fixed, but the systems developer gets to make some of the performance/resource usage (transistors) trade-off decisions. These decisions can involve selecting the word length, number of registers, and selecting between various possible instructions.^[142] The cost of retargeting a translator to such program-specific ASIPs has to be very low to make it worthwhile. Processor description driven code generators are starting to appear,^[260] which take the description used to specify the processor characteristics and build a translator for it. While the market for ASICs exceeds \$10 billion a year, the ASIP market is relatively small (but growing).
- *Number crunchers*. The quest for ever-more performance has led to a variety of designs that attempt to spread the load over more than one processor. Technical problems associated with finding sufficient work, in existing source code (which tends to have a serial rather than parallel form) to spread over more than one processor has limited the commercial viability of such designs. They have only proven cost effective in certain, application-specific domains where the computations have a natural mapping to multiple processors. The cost of the processor is often a significant percentage of the complete computing device. The market is small and the customers are likely to be individually known to the vendor.^[16] The use of clusters of low-price processors, as used in Beowulf, could see the demise of processors specifically designed for this market.^[39]

There are differences in processor characteristics within the domains just described. Processor design evolves over time and different vendors make different choices about the best way to use available resources (on chip transistors). For a detailed analysis of the issues involved for the Sun UltraSPARC processor, see.^[503]

The profile of the kinds of instructions generated for different processors can differ in both their static and their dynamic characteristics, even within the same domain. This was shown quite dramatically by Davidson, Rabung, and Whalley^[94] who measured static and dynamic instruction frequencies for nine different processors using the same translator (generating code for the different processors) on the same source files (see Figure 0.5). For a comparison of RISC processor instruction counts, based on the SPEC benchmarks, see McMahan and Lee.^[288]

The following are the lessons to be learned from the later discussions on processor details:

- Source code that makes the best use of one particular processor is unlikely to make the best use of any other processor.
- Making the best use of a particular processor requires knowledge of how it works and measurements of the program running on it. Without the feedback provided by the measurement of dynamic program

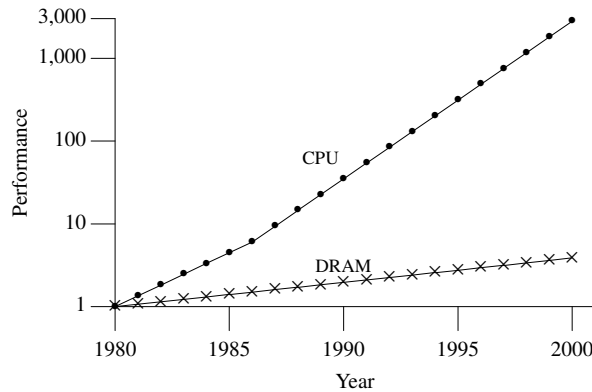


Figure 0.6: Relative performance of CPU against storage (DRAM), 1980==1. Adapted from Hennessy.^[163]

behavior, it is almost impossible to tune a program to any host.

5.1.1 Overcoming performance bottlenecks

There continues to be a market for processors that execute programs more quickly than those currently available. There is a commercial incentive to build higher-performance processors. Processor design has reached the stage where simply increasing the processor clock rate does not increase rate of program execution.^[384] A processor contains a variety of units, any one of which could be the bottleneck that prevents other units from delivering full performance. Some of these bottlenecks, and their solutions, can have implications at the source code level (less than perfect branch predictions^[208]) and others don't (the possibility of there being insufficient pins to support the data bandwidth required; pin count has only been increasing at 16% per year^[561]).

Data and instructions have to be delivered to the processor, from storage, to keep up with the rate it handles them. Using faster memory chips to keep up with the faster processors is not usually cost effective. Figure 0.6 shows how processor performance has outstripped that of DRAM (the most common kind of storage used). See Dietz and Mattox^[100] for measurements of access times to elements of arrays of various sizes, for 13 different Intel x86 compatible processors whose clock rates ranged from 100 MHz to 1700 MHz.

A detailed analysis by Agarwal, Hrishikesh, Keckler, and Burger^[3] found that delays caused by the time taken for signals to travel through on-chip wires (12–32 cycles to travel the length of a chip using 35nm CMOS technology, clocked at 10GHz), rather than transistor switching speed, was likely to be a major performance factor in future processors. Various methods have been proposed^[317] to get around this problem, but until such processor designs become available in commercially significant quantities they are not considered further here.

Cache

A commonly used solution to the significant performance difference between a processor and its storage is to place a small amount of faster storage, a *cache*, between them. Caching works because of locality of reference. Having accessed storage location X, a program is very likely to access a location close to X in the very near future. Research has shown^[163] that even with a relatively small cache (i.e., a few thousand bytes) it is possible to obtain significant reductions in accesses to main storage.

Modern, performance-based processors have two or more caches. A level 1 cache (called the *L1 cache*), which can respond within a few clock cycles (two on the Pentium 4, four on the UltraSPARC III), but is relatively small (8 K on the Pentium 4, 64 K on the UltraSPARC III), and a level 2 cache (called the *L2 cache*) which is larger but not as quick (256 K/7 clocks on the Pentium 4). Only a few processors have further levels of cache. Main storage is significantly larger, but its contents are likely to be more than 250 clock cycles away.

cache

Developer optimization of memory access performance is simplest when targeting processors that contain a cache, because the hardware handles most of the details. However, there are still cases where developers may need to manually tune memory access performance (e.g., application domains where large, sophisticated hardware caches are too expensive, or where customers are willing to pay for their applications to execute as fast as possible on their existing equipment). Cache behavior when a processor is executing more than one program at the same time can be quite complex.^[67,456]

The locality of reference used by a cache applies to both instructions and data. To maximize locality of reference, translators need to organize instructions in the order that an executing program is most likely to need them and allocate object storage so that accesses to them always fill the cache with values that will be needed next. Knowing which machine code sequences are most frequently executed requires execution profiling information on a program. Obtaining this information requires effort by the developer. It is necessary to instrument and execute a program on a representative set of data. This data, along with the original source is used by some translators to create a program image having a better locality of reference. It is also possible to be translator-independent by profiling and reorganizing the basic blocks contained in executable programs. Tomiyama and Yasuura^[455] used linear programming to optimize the layout of basic blocks in storage and significantly increased the instruction cache hit rate. Running as a separate pass after translation also reduces the need for interactive response times; the analysis took more than 10 hours on a 85 MHz microSPARC-II.

Is the use of a cache by the host processor something that developers need to take into account? Although every effort has been made by processor vendors to maximize cache performance and translator vendors are starting to provide the option to automatically tune the generated code based on profiling information,^[160] sometimes manual changes to the source (by developers) can make a significant difference. It is important to remember that any changes made to the source may only make any significant changes for one particular processor implementation. Other implementations within a processor family may share the same instruction set but they could have different cache behaviors. Cache-related performance issues are even starting to make it into the undergraduate teaching curriculum.^[248]

A comparison by Bahar, Calder, and Grunwald^[27] showed that code placement by a translator could improve performance more than a hardware-only solution; the two combined can do even better. In some cases the optimizations performed by a translator can affect cache behavior, for instance loop unrolling. Translators that perform such optimizations are starting to become commercially available.^[77] The importance of techniques for tuning specific kinds of applications are starting to be recognized (transaction processing as in Figure 0.8,^[5] numerical computations^[457]).

Specific cases of how optimizers attempt to maximize the benefits provided by a processor's cache are discussed in the relevant C sentences. In practice these tend to be reorganizations of the sequence of instructions executed, not reorganizations of the data structures used. Intel^[179] provides an example of how reorganization of a data structure can improve performance on the Pentium 4:

```

1  struct {
2      float x, y, z,   r, g, b;
3      } a_screen_3D[1000];
4  struct {
5      float x[1000], y[1000], z[1000];
6      float r[1000], g[1000], b[1000];
7      } b_screen_3D;
8  struct {
9      float x[4], y[4], z[4];
10     float r[4], g[4], b[4];
11     } c_screen_3D[250];

```

The structure declaration used for `a_screen_3D` might seem the obvious choice. However, it is likely that operations will involve either the tuple `x, y, and z`, or the tuple `r, g, and b`. A cache line on the Pentium 4 is 64 bytes wide, so a fetch of one of the `x` elements will cause the corresponding `r, g, and b` elements to be loaded. This is a waste of resource usage if they are not accessed. It is likely that all elements of the

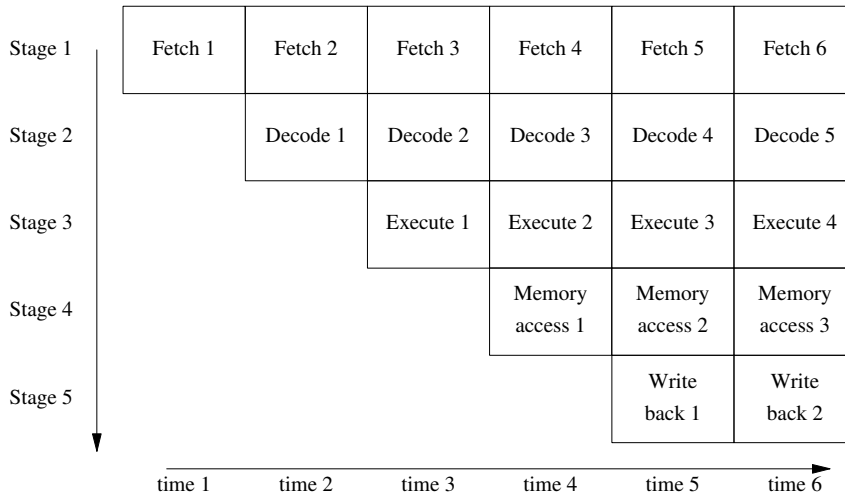


Figure 0.7: Simplified diagram of some typical stages in a processor instruction pipeline: Instruction fetch, decode, execute, memory access, and write back.

array will be accessed in sequence and the structure declaration used for `b_screen_3D` makes use of this algorithmic information. An access to an element of `x` will cause subsequent elements to be loaded into the cache line, ready for the next iteration. The structure declaration, suggested by Intel, for `c_screen_3D` makes use of a Pentium 4 specific characteristic; reading/writing 16 bytes from/to 16-byte aligned storage is the most efficient way to use the storage pipeline. Intel points to a possible 10% to 20% performance improvement through modifications that optimize cache usage; a sufficiently large improvement to warrant using the nonobvious, possibly more complex, data structures in some competitive markets.

Dividing up storage

Many host operating systems provide the ability for programs to make use of more storage than the host has physical memory (so-called *virtual memory*). This virtual memory is divided up into units called *pages*, which can be *swapped* out of memory to disk when it is not needed.^[163] There is a severe performance penalty on accesses to data that has been swapped out to disk (i.e., some other page needs to be swapped out and the page holding the required data items swapped back into memory from disk). Developers can organize data accesses to try to minimize this penalty. Having translators do this automatically, or even having them insert code into the program image to perform the swapping at points that are known to be not time-critical is a simpler solution.^[311]

storage
dividing up

Speeding up instruction execution

A variety of techniques are used to increase the number of instructions executed per second. Most processors are capable of executing more than one instruction at the same time. The most common technique, and one that can affect program behavior, is instruction *pipelining*. Pipelining breaks instruction execution down into a series of stages (see Figure 0.7). Having a different instruction processed by each stage at the same time does not change the execution time of a single instruction. But it does increase the overall rate of instruction execution because an instruction can complete at the end of every processor cycle. Many processors break down the stages shown in Figure 0.7 even further. For instance, the Intel Pentium 4 has a 20-stage pipeline.

processor
pipeline

The presence of a pipeline can affect program execution, depending on processor behavior when an exception is raised during instruction execution. A discussion on this issue is given elsewhere.

signal in-
terrupt
abstract machine
processing

Other techniques for increasing the number of instructions executed per second include: *VLIW* (Very Long Instruction Word) in which multiple operations are encoded in one long instruction, and parallel execution in which a processor contains multiple instruction execution units.^[450] These techniques have no more direct

impact on program behavior than instruction pipelining. In practice translators have had difficulty finding long sequences of instructions that can be executed in some concurrent fashion. Some help (e.g., source code annotations) from the developer is still needed for these processors to approach peak performance.

5.2 Runtime library

An implementation's runtime library handles those parts of a program that are not directly translated to machine code. Calls to the functions contained in this library may occur in the source or be generated by a translator (i.e., to some internal routine to handle arithmetic operations on values of type **long long**). The runtime library may be able to perform the operation completely (e.g., the trigonometric functions) or may need to call other functions provided by the host environment (e.g., O/S function, not C implementation functions).

6 Measuring implementations

Although any number of different properties of an implementation might be measured, the most commonly measured is execution time performance of the generated program image. In an attempt to limit the number of factors influencing the results, various organizations have created sets of test programs—*benchmarks*—that are generally accepted within their domain. Some of these test programs are discussed below (SPEC, the Transaction Processing council, Embedded systems, Linpack, and DSPSTONE). In some application areas the size of the program image can be important, but there are no generally accepted benchmarks for comparing size of program image. The growth in sales of mobile phones and other hand-held devices has significantly increased the importance of minimizing the electrical energy consumed by a program (the energy consumption needs of different programs performing the same function are starting to be compared^[36]).

A good benchmark will both mimic the characteristics of the applications it is intended to be representative of, and be large enough so that vendors cannot tune their products to perform well on it without also performing well on the real applications. The extent to which the existing benchmarks reflect realistic application usage is open to debate. Not only can different benchmarks give different results, but the same benchmark can exhibit different behavior with different input.^[105] Whatever their shortcomings may be the existing benchmarks are considered to be the best available (they are used in almost all published research).

It has long been an accepted truism that programs spend most of their time within loops and in particular a small number of such loops. Traditionally most processor-intensive applications, that were commercially important, have been scientific or engineering based. A third kind of application domain has now become commercially more important (in terms of hardware vendors making sales)—data-oriented applications such as transaction processing and data mining.

Some data-oriented applications share a characteristic with scientific and engineering applications in that a large proportion of their time is spent executing a small percentage of the code. However, it has been found that for Online Transaction Processing (OLTP), specifically the TPC-B benchmarks, the situation is more complicated.^[374] Recent measurements of four commercial databases running on an Intel Pentium processor showed that the processor spends 60% of its time stalled^[5] (see Figure 0.8).

A distinction needs to be made between characteristics that are perceived, by developers, to make a difference and those that actually do make a difference to the behavior of a program. Discussion within these Common Implementation sections is concerned with constructs that have been shown, by measurement, to make a difference to the execution time behavior of a program. Characteristics that relate to perceived differences fall within the realm of discussions that occur in the Coding guideline sections.

The measurements given in the Common Implementation sections tend to be derived from the characteristics of a program while it is being executed—dynamic measurements. The measurements given in the Usage sections tend to be based on what appears in the source code—static measurements.

6.1 SPEC benchmarks

Processor performance based on the SPEC (Standard Performance Evaluation Corporation, <http://www.spec.org>) benchmarks are frequently quoted by processor and implementation vendors. Academic research on optimizers often base their performance evaluations on the programs in the SPEC suite. SPEC benchmarks

Measuring imple-
mentations
program
image

iteration
statement
syntax

TPC-B 0

SPEC
benchmarks

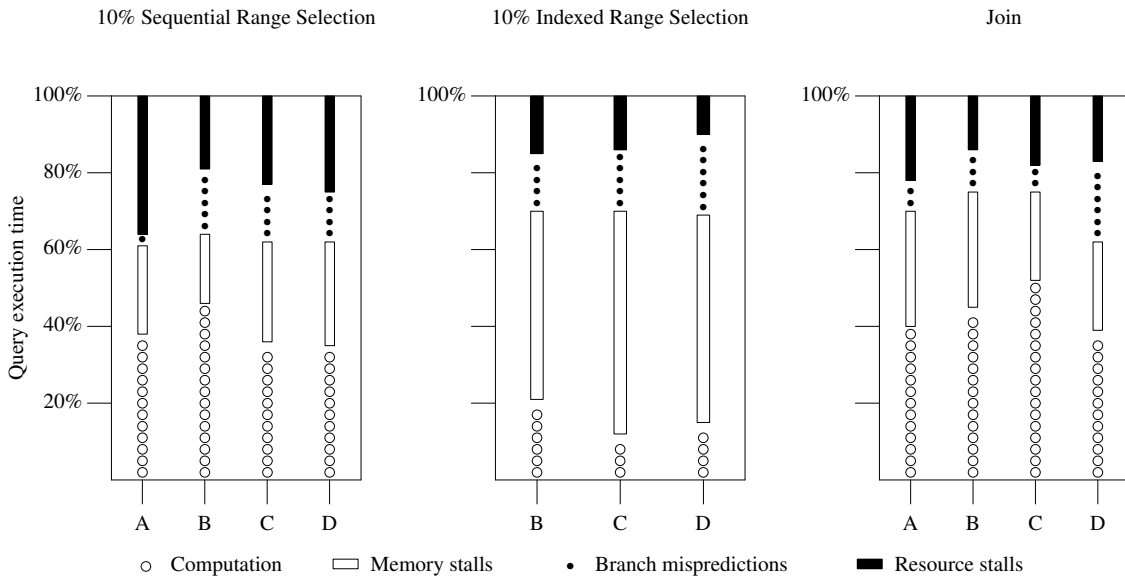


Figure 0.8: Execution time breakdown, by four processor components (bottom of graphs) for three different application queries (top of graphs). Adapted from Ailamaki.^[5]

cover a wide range of performance evaluations: graphics, NFS, mailservers, and CPU.^[102] The CPU benchmarks are the ones frequently used for processor and translator measurements.

The SPEC CPU benchmarks are broken down into two groups, the integer and the floating-point programs; these benchmarks have been revised over the years, the major releases being in 1989, 1992, 1995, and 2000. A particular set of programs is usually denoted by combining the names of these components. For instance, SPECint95 is the 1995 integer SPEC benchmark and SPECfp2000 is the 2000 floating-point benchmark.

The SPEC CPU benchmarks are based on publicly available source code (written in C for the integer benchmarks and, predominantly, Fortran and C for the floating-point). The names of the programs are known and versions of the source code are available on the Internet. The actual source code used by SPEC may differ slightly because of the need to be able to build and execute identical programs on a wide range of platforms (any changes needed to a program's source to enable it to be built are agreed to by the SPEC membership).

A study by Saavedra and Smith^[389] investigated correlations between constructs appearing in the source code and execution time performance of benchmarks that included SPEC.

6.2 Other benchmarks

The SPEC CPU benchmarks had their origins in the Unix market. As such they were and continue to be aimed at desktop and workstation platforms. Other benchmarks that are often encountered, and the rationale used in their design, include the following:

- DSPSTONE^[469] is a DSP-oriented set of benchmarks,
- The characteristics of programs written for embedded applications are very different.^[111] The EDN Embedded Microprocessor Benchmarking Consortium (EEMBC, pronounced Embassy — <http://www.eembc.org>), was formed in 1997 to develop standard performance benchmarks for the embedded market (e.g., telecommunications, automotive, networking, consumer, and office equipment). They currently have over 40 members and their benchmark results are starting to become known.
- MediaBench^[249] is a set of benchmarks targeted at a particular kind of embedded application—multimedia and communications. It includes programs that process data in various formats, including JPEG, MPEG, GSM, and postscript.

benchmarks

Olden benchmark	<ul style="list-style-type: none"> • The Olden benchmark^[60] attempts to measure the performance of architectures based on a distributed memory. • The Stanford Parallel Applications for SHared memory (SPLASH, now in its second release as SPLASH-2^[499]), is a suite of parallel applications intended to facilitate the study of centralized and distributed shared-address-space multiprocessors.
TPC-B	<ul style="list-style-type: none"> • The TPC-B benchmark from the Transaction Processing Performance Council (TPC).
Ranganathan ^[374]	<p><i>TPC-B models a banking database system that keeps track of customers' account balances, as well as balances per branch and teller. Each transaction updates a randomly chosen account balance, which includes updating the balance of the branch the customer belongs to and the teller from which the transaction is submitted. It also adds an entry to the history table which keeps a record of all submitted transactions.</i></p>

6.3 Processor measurements

Processor vendors also measure the characteristics of executing programs. Their reason is to gain insights that will enable them to build better products, either faster versions of existing processors or new processors. What are these measurements based on? The instructions executed by a processor are generated by translators, which may or may not be doing their best with the source they are presented with. Translator vendors may, or may not, have tuned their output to target processors with known characteristics. Fortunately this book does not need to concern itself further with this problem.

Processor measurements have been used to compare different processors,^[76] predict how many instructions a processor might be able to issue at the same time,^[420] and tune arithmetic operations.^[275] Processor vendors are not limited to using benchmarks or having access to source code to obtain useful information; Lee^[250] measured the instruction characteristics of several well-known Windows NT applications.

Coding Guidelines

7 Introduction

coding guidelines
introduction

The intent of these coding guidelines is to help management minimize the cost of ownership of the source code they are responsible for. The guidelines take the form of prepackaged recommendations of which source constructs to use, or not use, when more than one option is available. These coding guidelines sit at the bottom layer of what is potentially a complex, integrated software development environment.

Adhering to a coding guideline is an immediate cost. The discussion in these coding guidelines' sections is intended to help ensure that this cost payment is a wise investment yielding savings later.

The discussion in this section provides the background material for what appears in other coding guideline sections. It is also the longest section of the book and considers the following:

- The financial aspects of software development and getting the most out of any investment in adhering to coding guidelines.
- Selecting, adhering to, and deviating from guidelines.
- Applications and their influence on the source that needs to be written.
- Developers; bounding the limits, biases, and idiosyncrasies of their performance.

There are other Coding guideline subsections containing lengthy discussions. Whenever possible such discussions have been integrated into the C sentence-based structure of this book (i.e., they occur in the relevant C sentences).

The term used in this book to describe people whose jobs involve writing source code is *software developer*. The term *programmer* tends to be associated with somebody whose only job function is to write software. A

typist might spend almost 100% of the day typing. People do not spend all their time directly working on source code (in most studies, the time measured on this activity rarely rises above 25%), therefore the term programmer is not appropriate. The term software developer, usually shortened to *developer*, was chosen because it is relatively neutral, but is suggestive of somebody whose primary job function involves working with source code.

Developers often object to following coding guidelines, which are often viewed as restricting creative freedom, or forcing them to write code in some unnatural way. Creative freedom is not something that should be required at the source code implementation level. While particular ways of doing things may appeal to individual developers, such usage can be counter-productive. The cost to the original developer may be small, but the cost to subsequent developers (through requiring more effort by them to work with code written that way) may not be so small.

8 Source code cost drivers

Having source code occupy disk space rarely costs very much. The cost of ownership for source code is incurred when it is used. Possible uses of source code include: coding guidelines
cost drivers

- modifications to accommodate customer requests which can include fixing faults;
- major updates to create new versions of a product; and
- ports to new platforms, which can include new versions of platforms already supported.

These coding guideline subsections are applicable during initial implementation and subsequent modifications at the source code level. They do not get involved in software design issues, to the extent that these are programming language-independent. The following are the underlying factors behind these cost drivers:

- *Developer characteristics (human factors)*. Developers fail to deduce the behavior of source code constructs, either through ignorance of C or because of the limits in human information processing (e.g., poor memory of previously read code, perception problems leading to identifiers being misread, or information overload in short-term memory) causing faults to be introduced. These issues are dealt with here in the Coding guideline subsections.
- *Translator characteristics*. A change of translator can result in a change of behavior. Changes can include using a later version of the translator originally used, or a translator from a different vendor. Standards are rarely set in stone and the C Standard is certainly not. Variations in implementation behavior permitted by the standard means that the same source code can produce different results. Even the same translator can have its behavior altered by setting different options, or by a newer release. Differences in translator behavior are discussed in Commentary and Common Implementations subsections. Portability to C++ and C90 translators is dealt with in their respective sections.
- *Host characteristics*. Just like translator behavior this can vary between releases (updates to system libraries) and host vendors. The differences usually impact the behavior of library calls, not the language. These issues are dealt with in Common Implementation sections.
- *Application characteristics*. Programs vary in the extent to which they need to concern themselves with the host on which they execute— for instance, accessing memory ports. They can also place different demands on language constructs— for instance, floating-point or dynamic memory allocation. These issues are dealt with under Usage, indirectly under Common Implementations and here in Coding Usage
1 Guideline sections.
- *Product testing*. The complexity of source code can influence the number of test cases that need to be written and executed. This complexity can be affected by design, algorithmic and source code construct selection issues. The latter can be the subject of coding guidelines. coding
guidelines
testability

Covering all possible source code issues is impossible. Frequency of occurrence is used to provide a cutoff filter. The main purpose of the information in the Usage sections is to help provide evidence for what filtering Usage
1 to apply.

8.1 Guideline cost/benefit

coding guidelines
importance

When a guideline is first encountered it is educational. It teaches developers about a specific problem that others have encountered and that they are likely to encounter. This is a one-time learning cost (that developers are likely to have to pay at some time in their careers). People do forget, so there may be a relearning cost. (These oversights are the sort of thing picked up by an automated guideline enforcement tool, jogging the developer's memory in the process.)

NPV 0

Adhering to guidelines requires an investment in the form of developer's time. Like all investments it needs to be made on the basis that a later benefit will provide an adequate return. It is important to bear in mind that failure to recoup the original investment is not the worst that can happen. The value of lost opportunity through being late to market with a product can equal the entire development budget. It is management's responsibility to select those coding guidelines that have a return on investment applicable to a particular project.

A set of guidelines can be viewed as a list of recommended coding practices, the economic cost/benefit of which has been precalculated and found to be acceptable. This precalculation, ideally, removes the need for developers to invest in performing their own calculations. (Even in many situations where they are not worthwhile, the cost of performing the analysis is greater than the cost of following the guideline.)

Researchers^[31,439] are only just starting to attempt to formally investigate the trade-off involved between the cost of creating maintainable software and the cost of maintaining software.

A study by Visaggio^[481] performed a retrospective analysis of a reengineering process that had been applied to a legacy system containing 1.5 M lines. The following is his stated aim:

Visaggio^[481]

1. Guidelines are provided for calculating the quality and economic scores for each component; These can be reused in other projects, although they can and must also be continually refined with use;
2. A model for determining the thresholds on each axis is defined; the model depends on the quality and economics policy adopted by the organization intending to renew the legacy system;
3. A decision process is included, that helps to establish which renewal process should be carried out for each component; this process may differ for components belonging to the same quadrant and depends on the targets the organization intends to attain with the renewal process.

8.1.1 What is the cost?

coding guidelines
the cost

Guidelines may be more or less costly to follow (in terms of modifying, or not using, constructs once their lack of conformance to a guideline is known). Estimating any cost change caused by having to use constructs not prohibited by a guideline will vary from case to case. It is recognized that the costs of following a guideline recommendation can be very high in some cases. One solution is the deviation mechanism, which is discussed elsewhere.

deviations 0
coding guidelines

Guidelines may be more or less easy to flag reliably from a static analysis tool point of view. The quality of static analysis tools is something that developers need to evaluate when making a purchase decision. These coding guidelines recognize the difficulties in automating some checks by indicating that some should be performed as part of code reviews.

code reviews 0

All guidelines are given equal weight in terms of the likelihood of not adhering to them causing a fault. Without data correlating a guideline not being followed to the probability of the containing code causing a fault, no other meaningful options are available.

8.1.2 What is the benefit?

coding guidelines
the benefit

What is the nature of the benefit obtained from an investment in adhering to coding guidelines? These coding guidelines assume that the intended final benefit is always financial. However, the investment proposal may not list financial benefits as the immediate reason for making it. Possible other reasons include:

- mandated by some body (e.g., regulatory authority, customer Q/A department);

- legal reasons— companies want to show that they have used industry best practices, whatever they are, in the event of legal action being taken against them;
- a mechanism for controlling source code: The purpose of this control may be to reduce the dependency on a particular vendor's implementation (portability issues), or it may be an attempt to overcome inadequacies in developer training.

Preventing a fault from occurring is a benefit. How big is this benefit (i.e., what would the cost of the fault have been? How is the cost of a fault measured?) Is it in terms of the cost of the impact on the end user of experiencing the fault in the program, or is it the cost to the vendor of having to deal with it being uncovered by their customers (which may include fixing it)? Measuring the cost to the end user is very difficult to do, and it may involve questions that vendors would rather have left unasked. To simplify matters these guidelines are written from the point of view of the vendor of the product containing software. The cost we consider is the cost to fix the fault multiplied by the probability of the fault needing to be fixed (fault is found and customer requirements demand a fix).

8.1.3 Safer software?

Coding guidelines, such as those given in this book, are often promoted as part of the package of measures to be used during the development of safety-critical software.

coding guidelines
safer software

The fact that adherence to guideline recommendations may reduce the number of faults introduced into the source by developers is primarily an economic issue. The only difference between safety critical software and other kinds of software is the level of confidence required that a program will behave as intended. Achieving a higher level of confidence often involves a higher level of cost. While adherence to guideline recommendations may reduce costs and enable more confidence level boosting tasks to be performed, for the same total cost, management may instead choose to reduce costs and not perform any additional tasks.

Claiming that adhering to coding guidelines makes programs safer suggests that the acceptance criteria being used are not sufficient to achieve the desired level of confidence on their own (i.e., reliance is being placed on adherence to guideline recommendations reducing the probability of faults occurring in sections of code that have not been fully tested).

An often-heard argument is that some language constructs are the root cause of many faults in programs and that banning the use of these constructs leads to fewer faults. While banning the use of these constructs may prevent them from being the root cause of faults, there is rarely any proof that the alternative constructs used will not introduce as many faults, if not more, than the constructs they replace.

This book does not treat *safety-critical* as being a benefit of adherence to guideline recommendations in its own right.

8.2 Code development's place in the universe

Coding guidelines need to take account of the environment in which they will be applied. There are a variety of reasons for creating programs. Making a profit is a common rationale and the only one considered by these coding guidelines. Writing programs for enjoyment, by individuals, involves reasons of a personal nature, which are not considered in this book.

development
context

A program is created by developers who will have a multitude of reasons for doing what they do. Training and motivating these developers to align their interests with that of the organization that employs them is outside the scope of this book, although staffing issues are discussed.

coding
guidelines
staffing

Programs do not exist in isolation. While all applications will want fault-free software, the importance assigned to faults can depend on the relative importance of the software component of the total package. This relative importance will also influence the percentage of resources assigned to software development and the ability of the software manager to influence project time scales.

The kind of customers an organization sells to, can influence the software development process. There are situations where effectively there is a single customer. For instance, a large organization paying for the development of a bespoke application will invariably go through a formal requirements analysis, specification,

design, code, test, and handover procedure. Much of the research on software development practices has been funded by and for such development projects. Another example is software that is invisible to the end user, but is part of a larger product. Companies and projects differ as to whether software controls the hardware or vice versa (the hardware group then being the customer).

Most *Open Source* software development has a single customer, the author of the software.^[141,322] In this case the procedures followed are likely to be completely different from those followed by paying customers. In a few cases Open Source projects involving many developers have flourished. Several studies^[304] have investigated some of the group dynamics of such cooperative development (where the customer seems to be the members of a core team of developers working on the project). While the impact of this form of production on traditional economic structures is widely thought to be significant,^[41] these guidelines still treat it as a form of production, which has different cost/benefit cost drivers; whether the motivating factors for individual developers are really any different is not discussed here.

When there are many customers, costs are recouped over many customers, who usually pay less than the development cost of the software. In a few cases premium prices can be charged by market leaders, or by offering substantial customer support. The process used for development is not normally visible to the customer. Development tends to be led by marketing and is rarely structured in any meaningful formal way; in fact too formal a process could actively get in the way of releasing new products in a timely fashion.

Research by Carmel^[62] of 12 firms (five selling into the mass market, seven making narrow/direct sales) involved in packaged software development showed that the average firm has been in business for three years, employed 20 people, and had revenues of \$1 million (1995 figures).

As pointed out by Carmel and others, time to market in a competitive environment can be crucial. Being first to market is often a significant advantage. A vendor that is first, even with a very poorly architected, internally, application often gets to prosper. While there may be costs to pay later, at least the company is still in business. A later market entrant may have a wonderfully architected product that has scope for future expansion and minimizes future maintenance costs, but without customers it has no future.

A fundamental problem facing software process improvement is how best to allocate limited resources, to obtain optimal results. Large-scale systems undergo continuous enhancement and subcontractors may be called in for periods of time. There are often relatively short release intervals and a fixed amount of resources. These characteristics prohibit revolutionary changes to a system. Improvements have to be made in an evolutionary fashion.

Coding guidelines need to be adaptable to these different development environments. Recognizing that guideline recommendations will be adapted, it is important that information on the interrelationship between them is made available to the manager. These interrelationships need to be taken into account when tailoring a set of guideline recommendations.

8.3 Staffing

The culture of information technology appears to be one of high staff turnover^[310] (with reported annual turnover rates of 25% to 35% in Fortune 500 companies).

If developers cannot be retained on a project new ones need to be recruited. There are generally more vacancies than there are qualified developers to fill them. Hiring staff who are less qualified, either in application-domain knowledge or programming skill, often occurs (either through a conscious decision process or because the developer's actual qualifications were not appreciated). The likely competence of future development staff may need to be factored into the acceptable complexity of source code.

A regular turnover of staff creates the need for software that does not require a large investment in upfront training costs. While developers do need to be familiar with the source they are to work on, companies want to minimize familiarization costs for new staff while maximizing their productivity. Source code level guideline recommendations can help reduce familiarization costs in several ways:

- Not using constructs whose behavior varies across translator implementations means that recruitment does not have to target developers with specific implementation experience, or to factor in the cost of

retraining—it will occur, usually through on-the-job learning.

- Minimizing source complexity helps reduce the cognitive effort required from developers trying to comprehend it.
- Increased source code memorability can reduce the number of times developers need to reread the same source.
- Visible source code that follows a consistent set of idioms can take advantage of people’s natural ability to categorize and make deductions based on these categorizes.

Implementing a new project is seen, by developers, as being much more interesting and rewarding than maintaining existing software. It is common for the members of the original software to move on to other projects once the one they are working is initially completed. Studies by Couger and Colter^[84] investigated various approaches to motivating developers working on maintenance activities. They identified the following two factors:

1. *The motivating potential of the job*, based on skill variety required, the degree to which the job requires completion as a whole (task identity), the impact of the job on others (task significance), degree of freedom in scheduling and performing the job, and feedback from the job (used to calculate a *Motivating Potential Score*, MPS).
2. *What they called an individual’s growth need strength (GNS)*, based on a person’s need for personal accomplishment, to be stimulated and challenged.

The research provided support for the claim that MPS and GNS could be measured and that jobs could be tailored, to some degree, to people. Management’s role was to organize the work that needed to be done so as to balance the MPS of jobs against the GNS of the staff available.

It is your author’s experience that very few companies use any formally verified method for measuring developer characteristics, or fitting their skills to the work that needs to be done. Project staffing is often based on nothing more than staff availability and a date by which the tasks must be completed.

8.3.1 Training new staff

Developers new to a project often need to spend a significant amount of time (often months) building up their knowledge base of a program’s source code.^[411] One solution is a training program, complete with well-documented introductions, road maps of programs, and how they map to the application domain, all taught by well-trained teachers. While this investment is cost effective if large numbers of people are involved, most source code is worked on by a relatively small number of people. Also most applications evolve over time. Keeping the material up-to-date could be difficult and costly, if not completely impractical. In short, the cost exceeds the benefit.

developer
training

In practice new staff have to learn directly from the source code. This may be supplemented by documentation, provided it is reasonably up-to-date. Other experienced developers who have worked on the source may also be available for consultation.

8.4 Return on investment

The risk of investing in the production of software is undertaken in the expectation of receiving a return that is larger than the investment. Economists have produced various models that provide an answer for the question: “What return should I expect from investing so much money at such and such risk over a period of time?”

ROI

Obtaining reliable estimates of the risk factors, the size of the financial investment, and the time required is known to be very difficult. Thankfully, they are outside the scope of this book. However, given the prevailing situation within most development groups, where nobody has had any systematic cost/benefit analysis training, an appreciation of the factors involved can provide some useful background.

Minimizing the total cost of a software product (e.g., balancing the initial development costs against subsequent maintenance costs) requires that its useful life be known. The risk factors introduced by third parties (e.g., competitive products may remove the need for continued development, customers may not purchase the product) mean that there is the possibility that any investment made during development will never be realized during maintenance because further work on the product never occurs.

The physical process of writing source code is considered to be so sufficiently unimportant that doubling the effort involved is likely to have a minor impact on development costs. This is the opposite case to how most developers view the writing process. It is not uncommon for developers to go to great lengths to reduce the effort needed during the writing process, paying little attention to subsequent effects of their actions; reports have even been published on the subject.^[367]

8.4.1 Some economics background

Before going on to discuss some of the economic aspects of coding guidelines, we need to cover some of the basic ideas used in economics calculations. The primary quantity that is used in this book is Net Present Value (NPV).

8.4.1.1 Discounting for time

A dollar today is worth more than a dollar tomorrow. This is because today's dollar can be invested and start earning interest immediately. By tomorrow it will have increased in value. The present value (PV) of a future payoff, C , can be calculated from:

$$PV = \text{discount factor} \times C \quad (0.3)$$

where the *discount factor* is less than one. It is usually represented by:

$$\text{discount factor} = \frac{1}{1 + r} \quad (0.4)$$

where r is known as the rate of return; representing the amount of reward demanded by investors for accepting a delayed payment. The rate of return is often called the *discount rate* or the *opportunity cost* of capital. It is often quoted over a period of a year, and the calculation for PV over n years becomes:

$$PV = \frac{C}{(1 + r)^n} \quad (0.5)$$

By expressing all future payoffs in terms of present value, it is possible to compare them on an equal footing.

Example (from Raffo^[372]). A manager has the choice of spending \$250,000 on the purchase of a test tool, or the same amount of money on hiring testers. It is expected that the tool will make an immediate cost saving of \$500,000 (by automating various test procedures). Hiring the testers will result in a saving of \$750,000 in two years time. Which is the better investment (assuming a 10% discount rate)?

$$PV_{\text{tool}} = \frac{\$500,000}{(1 + 0.10)^0} = \$500,000 \quad (0.6)$$

$$PV_{\text{testers}} = \frac{\$750,000}{(1 + 0.10)^2} = \$619,835 \quad (0.7)$$

Based on these calculations, hiring the testers is the better option (has the greatest present value).

8.4.1.2 Taking risk into account

The previous example did not take risk into account. What if the tool did not perform as expected, what if some of the testers were not as productive as hoped? A more realistic calculation of present value needs to take the risk of future payoffs not occurring as expected into account.

A risky future payoff is not worth as much as a certain future payoff. The risk is factored into the discount rate to create an *effective discount rate*: $k = r + \theta$ (where r is the risk-free rate and θ a premium that depends on the amount of risk). The formulae for present value becomes:

$$PV = \frac{C}{1 + k^n} \quad (0.8)$$

Recognizing that both r and θ can vary over time we get:

$$PV = \sum_{i=1}^t \frac{\text{return}_i}{1 + k_i} \quad (0.9)$$

where return_i is the return during period i .

Example. Repeating the preceding example, but assuming a 15% risk premium for the testers option.

$$PV_{\text{tool}} = \frac{\$500,000}{(1 + 0.10)^0} = \$500,000 \quad (0.10)$$

$$PV_{\text{testers}} = \frac{\$750,000}{(1 + 0.10 + 0.15)^2} = \$480,000 \quad (0.11)$$

Taking this risk into account shows that buying the test tool is the better option.

8.4.1.3 Net Present Value

Future payoffs do not just occur, an investment needs to be made. A quantity called the *Net Present Value* (NPV) is generally considered to lead to the better investment decisions.^[52] It is calculated as:

$$NPV = PV - \text{investment cost} \quad (0.12)$$

Example (from Raffo^[372]). A coding reading initiative is expected to cost \$50,000 to implement. The expected payoff, in two years time, is \$100,000. Assuming a discount rate of 10%, we get:

$$NPV = \frac{\$100,000}{1 \times 10^2} - \$50,000 = \$32,645 \quad (0.13)$$

Several alternatives to NPV, their advantages and disadvantages, are described in Chapter five of Brealey^[52] and by Raffo.^[372] One commonly seen rule within rapidly changing environments is the payback rule. This requires that the investment costs of a project be recovered within a specified period. The payback period is the amount of time needed to recover investment costs. A shorter payback period being preferred to a longer one.

8.4.1.4 Estimating discount rate and risk

The formulae for calculating value are of no use unless reliable figures for the discount rate and the impact of risk are available. The discount rate represents the risk-free element and the closest thing to a risk-free investment is government bonds and securities. Information on these rates are freely available. Governments face something of a circularity problem in how they calculate the discount rate for their own investments. The US government discusses these issues in its *Guidelines and Discount Rates for Benefit-Cost Analysis of Federal Programs*^[489] and specifies a rate of 7%.

Analyzing risk is a much harder problem. Information on previous projects carried out within the company can offer some guidance on the likelihood of developers meeting productivity targets. In a broader context the market conditions also need to be taken into account, for instance: how likely is it that other companies will bring out competing products? Will demand for the application still be there once development is complete?

One way of handling these software development risks is for companies to treat these activities in the same way that options are used to control the risk in a portfolio of stocks. Some very sophisticated models and formula for balancing the risks involved in holding a range of assets (e.g., stocks) have been developed. The match is not perfect in that these methods rely on a liquid market, something that is difficult to achieve using people (moving them to a new project requires time for them to become productive). A number of researchers^[122,436,496] have started to analyze some of the ways these methods might be applied to creating and using options within the software development process.

8.5 Reusing software

It is rare for a single program to handle all the requirements of a complete application. An application is often made up of multiple programs and generally there is a high degree of similarity in many of the requirements for these programs. In other cases there may be variations in a hardware/software product. Writing code tailored to each program or product combination is expensive. Reusing versions of the same code in multiple programs sounds attractive.

In practice code reuse is a complex issue. How to identify the components that might be reusable, how much effort should be invested in writing the original source to make it easy to reuse, how costs and benefits should be apportioned are a few of the questions.

A survey of the economic issues involved in software reuse is provided by Wiles.^[491] These coding guidelines indirectly address code reuse in that they recommend against the use of constructs that can vary between translator implementations.

8.6 Using another language

A solution that is sometimes proposed to get around problems in C that are seen as the root cause of many faults is to use another language. Commonly proposed languages include Pascal, Ada, and recently Java. These languages are claimed to have characteristics, such as strong typing, that help catch faults early and reduce maintenance costs.

In 1987 the US Department of Defense mandated Ada (DoD Directive 3405.1) as the language in which bespoke applications, written for it, had to be written. The aim was to make major cost savings over the full lifetime of a project (implementation and maintenance, throughout its operational life); the higher costs of using Ada during implementation^[377] being recovered through reduced maintenance costs over its working lifetime.^[504] However, a crucial consideration had been overlooked in the original cost analysis. Many projects are canceled before they become operational.^[117,429] If the costs of all projects, canceled or operational, are taken into account, Ada is not the most cost-effective option. The additional cost incurred during development of projects that are canceled exceeds the savings made on projects that become operational. The directive mandating the use of Ada was canceled in 1997.^[468]

Proposals to use other languages sometimes have more obvious flaws in their arguments. An analysis of why Lisp should be used^[121] is based on how that language overcomes some of the C-inherent problems, while overlooking its own more substantial weaknesses (rather like proposing that people hop on one leg as a solution to wearing out two shoes by walking on two).

The inability to think through a reasoned argument, where choice of programming language is concerned, is not limited to academic papers^[318] (5.3.11 Safe Subsets of Programming languages).

The use of software in applications where there is the possibility of loss of life, or serious injury, is sometimes covered by regulations. These regulations often tend to be about process— making sure that various checks are carried out. But sometimes subsets of the C language have been defined (sometimes called by the name safe subsets). The associated coding guideline is that constructs outside this subset not be used. Proof for claiming that use of these subsets result in safer programs is nonexistent. The benefit of following coding guidelines is discussed elsewhere.

o coding
guidelines
the benefit

8.7 Testability

This subsection is to provide some background on testing programs. The purpose of testing is to achieve a measurable degree of confidence that a program will behave as expected. Beizer^[38] provides a practical introduction to testing.

coding guidelines
testability

Testing is often written about as if its purpose were to find faults in applications. Many authors quote figures for the cost of finding a fault, looking for cost-effective ways of finding them. This outlook can lead to an incorrectly structured development process. For instance, a perfect application will have an infinite cost per fault found, while a very badly written application will have a very low cost per fault found. Other figures often quoted involve the cost of finding faults in different phases of the development process. In particular, the fact that the cost per fault is higher, the later in the process it is discovered. This observation about relative costs often occurs purely because of how development costs are accounted for. On a significant development effort equipment and overhead costs tend to be fixed, and there is often a preassigned number of people working on a particular development phase. These costs are not likely to vary by much, whether there is a single fault found or 100 faults. However, it is likely that there will be significantly fewer faults found in later phases because most of them will have been located in earlier phases. Given the fixed costs that cannot be decreased, and the smaller number of faults, it is inevitable that the cost per fault will be higher in later phases.

Many of the faults that exist in source code are never encountered by users of an application. Examples of such faults are provided in a study by Chou, Yang, Chelf, Hallem, and Engler^[72] who investigated the history of faults in the Linux kernel (found using a variety of static analysis tools). The source of different releases of the Linux kernel is publicly available (for this analysis 21 snapshots of the source over a seven year period were used). The results showed how faults remained in successive releases of code that was used for production work in thousands (if not hundreds of thousands) of computers. The average fault lifetime, before being fixed, or the code containing it ceasing to exist was 1.8 years.

The following three events need to occur for a fault to become an application failure:

1. A program needs to execute the statement containing the fault.
2. The result of that execution needs to infect the subsequent data values, another part of the program.
3. The infected data values must propagate to the output.

The probability of a particular fault affecting the output of an application for a given input can be found by multiplying together the probability of the preceding three events occurring for that set of input values. The following example is taken from Voas:^[482]

```

1  #include <math.h>
2  #include <stdio.h>
3
4  void quadratic_root(int a, int b, int c)
5  /*
6   * If one exists print one integral solution of:
7   * ax^2 + bx + c = 0
8   */
9  {
```

```

10 int d,
11     x;
12
13 if (a != 0)
14     {
15     d = (b * b) - (5 * a * c); /* Fault, should multiply by 4. */
16     if (d < 0)
17         x = 0;
18     else
19         x = (sqrt(d) / (2 * a)) - b;
20     }
21 else
22     x = -(c / b);
23
24 if ((a * x * x + b * x + c) == 0)
25     printf("%d is an integral solution\n", x);
26 else
27     printf("There is no integral solution\n");
28 }

```

Execution of the function `quadratic_root` has four possibilities:

1. The fault is not executed (e.g., `quadratic_root(0, 3, 6)`).
2. The fault is executed but does not infect any of the data (e.g., `quadratic_root(3, 2, 0)`).
3. The fault is executed and the data is infected, but it does not affect the output (e.g., `quadratic_root(1, -1, -12)`).
4. The fault is executed and the infected data causes the output to be incorrect (e.g., `quadratic_root(10, 0, 10)`).

This program illustrates the often-seen situations of a program behaving as expected because the input values used were not sufficient to turn a fault in the source code into an application failure during program execution.

Testing by execution examines the source code in a different way than is addressed by these coding guidelines. One looks at only those parts of the program (in translated form) through which flow of control passes and applies specific values, the other examines source code in symbolic form.

A study by Adams^[2] looked at faults found in applications over time. The results showed (see Table 0.1) that approximately one third of all detected faults occurred on average every 5,000 years of execution time. Only around 2% of faults occurred every five years of execution time.

Table 0.1: Percentage of reported problems having a given mean time to first problem occurrence (in months, summed over all installations of a product) for various products (numbered 1 to 9), e.g., 28.8% of the reported faults in product 1 were, on average, first reported after 19,000 months of program execution time (another 34.2% of problems were first reported after 60,000 months). From Adams.^[2]

Product	19	60	190	600	1,900	6,000	19,000	60,000
1	0.7	1.2	2.1	5.0	10.3	17.8	28.8	34.2
2	0.7	1.5	3.2	4.5	9.7	18.2	28.0	34.3
3	0.4	1.4	2.8	6.5	8.7	18.0	28.5	33.7
4	0.1	0.3	2.0	4.4	11.9	18.7	28.5	34.2
5	0.7	1.4	2.9	4.4	9.4	18.4	28.5	34.2
6	0.3	0.8	2.1	5.0	11.5	20.1	28.2	32.0
7	0.6	1.4	2.7	4.5	9.9	18.5	28.5	34.0
8	1.1	1.4	2.7	6.5	11.1	18.4	27.1	31.9
9	0.0	0.5	1.9	5.6	12.8	20.4	27.6	31.2

8.8 Software metrics

In a variety of engineering disciplines, it is possible to predict, to within a degree of uncertainty, various behaviors and properties of components by measuring certain parameters and matching these measurements against known behaviors of previous components having similar measurements. A number of software metrics (software measurements does not sound as scientific) are based on the number of lines of source code. Comments are usually excluded from this count. What constitutes a line is at times fiercely debated.^[336] The most commonly used count is based on a simple line count, ignoring issues such as multiple statements on one line or statements spanning more than one line.

The results from measurements of software are an essential basis for any theoretical analysis.^[126] However, some of the questions people are trying to answer with measurements of source code have serious flaws in their justification. Two commonly asked questions are the effort needed to implement a program (before it is implemented) and the number of faults in a program (before it is shipped to customers). Fenton attempted to introduce a degree of rigour into the use of metrics.^[124, 125]

The COCOMO project (COConstructive COSt Model, the latest release is known as COCOMO II) is a research effort attempting to produce an Open Source, public domain, software cost, effort, and schedule for developing new software development. Off-the-shelf, untuned models have been up to 600% inaccurate in their estimates. After Bayesian tuning models that are within 30% of the actual figures 71% of the time have been built.^[73] Effort estimation is not the subject of this book and is not discussed further.

These attempts to find meaningful measures all have a common goal — the desire to predict. However, most existing metrics are based on regression analysis models, they are not causal models. To build these models, a number of factors believed to affect the final result are selected, and a regression analysis is performed to calculate a correlation between them and the final results. Models built in this way will depend on the data from which they were built and the factors chosen to correlate against. Unlike a causal model (which predicts results based on “telling the story”,^[124]) there is no underlying theory that explains how these factors interact. For a detailed critique of existing attempts at program defect prediction based on measures of source code and fault history, see Fenton.^[124]

The one factor that existing fault-prediction models ignore is the human brain/mind. The discussion in subsequent sections should convince the reader that source code complexity only exists in the mind of the reader. Without taking into account the properties in the reader’s mind, it is not possible to calculate a complexity value. For instance, one frequently referenced metric is Halstead’s software science metric, which uses the idea of the *volume* of a function. This *volume* is calculated by counting the operators and operands appearing in that function. There is no attempt to differentiate functions containing a few complex expressions from functions containing many simple expressions; provided the total and unique operand/operator count is the same, they will be assigned the same complexity.

9 Background to these coding guidelines

These coding guidelines are conventional, if a little longer than most, in the sense that they contain the usual exhortation not to use a construct, to do things a particular way, or to watch out for certain problems. They are unconventional because of the following:

- An attempt has been made to consider the impact of a prohibition— do the alternatives have worse cost/benefit?
- Deviations are suggested— experience has shown that requiring a yes/no decision on following a guideline recommendation can result in that recommendation being ignored completely. Suggesting deviations can lead to an increase in guideline recommendations being followed by providing a safety valve for the awkward cases.
- Economics is the only consideration— it is sometimes claimed that following guideline recommendations imbues software with properties such as being better or safer. Your author does not know of any way of measuring betterness in software. The case for increased safety is discussed elsewhere.

- An attempt has been made to base those guideline recommendations that relate to human factors on the experimental results and theories of cognitive psychology.

cognitive
psychology

The wording used in these guideline recommendations is short and to the point (and hopefully unambiguous). It does assume some degree of technical knowledge. There are several ISO standards^[182, 191] dealing with the wording used in the specification of a computer language. The principles of designing and documenting procedures to be carried out by others are thoroughly covered by Degani and Wiener.^[98]

It is all very well giving guideline recommendations for developers to follow. But, how do they do their job. How were they selected? When do they apply? These are the issues discussed in the following sections.

9.1 Culture, knowledge, and behavior

Every language has a culture associated with its use. A culture entails thinking about and doing certain things in a certain way.^[326] How and why these choices originally came about may provide some interesting historical context and might be discussed in other sections of this book, but they are generally not relevant to Coding guideline sections.

Culture is perhaps too grand a word for the common existing practices of C developers. Developers are overconfident and insular enough already without providing additional blankets to wrap themselves in. The term *existing practice* is both functional and reduces the possibility of aggrandizement.

Existing practices could be thought of as a set of assumptions and expectations about how things are done (in C). The term *C style* is sometimes used to describe these assumptions and expectations. However, this term has so many different meanings, for different developers, in different contexts, that its use is very prone to misunderstanding and argument. Therefore every effort will be made to stay away from the concept of style in this book.

coding
guidelines
coding style

In many ways existing practice is a *meme machine*.^[47] Developers read existing code, learn about the ideas it contains, and potentially use those ideas to write new code. Particular ways of writing code need not be useful to the program that contains them. They only need to appear to be useful to the developer who writes the code, or fit in with a developer's preferred way of doing things. In some cases developers do not thoroughly analyze what code to write, they follow the lead of others. Software development has its fads and fashions, just like any other information-driven endeavor.^[46]

Before looking at the effect of existing practice on coding guidelines we ought to ask what constitutes existing practice. As far as the guideline recommendations in this book are concerned, what constitutes existing practice is documented in the Usage subsections. Developers are unlikely to approach this issue in such a scientific way. They will have worked in one or more application domains, been exposed to a variety of source code, and discussed C with a variety of other developers. While some companies might choose to tune their guidelines to the practices that apply to specific application domains and working environments, the guideline recommendations in this book attempt to be generally applicable.

measure-
ments
SESS

Existing practices are not always documented and, in some cases, developers cannot even state what they are. Experienced developers sometimes use expressions such as *the C way of doing things*, or *I feel*. When asked what is meant by these expressions, they are unable to provide a coherent answer. This kind of human behavior (knowing something without being able to state what it is) has been duplicated in the laboratory.

implicit learning

- A study by Lewicki, Hill and Bizot^[262] demonstrated the effect of implicit learning on subjects expectations, even when performing a task that contained no overt learning component. In this study, while subjects watched a computer screen a letter was presented in one of four possible locations. Subjects had to press the button corresponding to the location of the letter as quickly as possible. The sequence of locations used followed a consistent, but complex, pattern. The results showed subjects' response times continually improving as they gained experience. The presentation was divided into 17 segments of 240 trials (a total of 4,080 letters), each segment was separated by a 10-second break. The pattern used to select the sequence of locations was changed after the 15th segment (subjects were not told about the existence of any patterns of behavior). When the pattern changed, the response times

immediately got worse. After completing the presentation subjects were interviewed to find out if they had been aware of any patterns in the presentation; they had not.

- A study by Reber and Kassir^[376] compared implicit and explicit pattern detection. Subjects were asked to memorize sets of words containing the letters *P*, *S*, *T*, *V*, or *X*. Most of these words had been generated using a finite state grammar. However, some of the sets contained words that had not been generated according to the rules of this grammar. One group of subjects thought they were taking part in a purely memory-based experiment; the other group was also told to memorize the words but was also told of the existence of a pattern in the letter sequences and that it would help them in the task if they could deduce this pattern. The performance of the group that had not been told about the presence of a pattern almost exactly mirrored that of the group who had been told on all sets of words (pattern words only, pattern plus non-pattern words, non-pattern words only). Without being told to do so, subjects had used patterns in the words to help perform the memorization task.
- A study carried out by Berry and Broadbent^[42] asked subjects to perform a task requiring decision making using numerical quantities. In these experiments subjects were told that they were in charge of a sugar production factory. They had to control the rate of sugar production to ensure it kept at the target rate of 9,000 tons. The only method of control available to them was changing the size of the workforce. Subjects were not told anything about the relationship between the current production rate, the number of workers and previous production rates. The starting point was 600 workers and an output rate of 6,000 tons. Subjects had to specify the number of workers they wished to employ and were then told the new rate of production (interaction was via a terminal connected to a computer). At the end of the experiment, subjects had to answer a questionnaire about the task they had just performed. The results showed that although subjects had quickly learned to keep the rate of sugar production close to the desired level, they were unable to verbalize how they achieved this goal.

letter patterns
implicit learning

The studies performed by these and other researchers demonstrate that it is possible for people to perform quite complex tasks using knowledge that they are not consciously aware of having. By working with other C developers and reading existing C source code, developers obtain the nonverbalized knowledge that is part of the unwritten culture of C. This knowledge is expressed by developers having expectations and making assumptions about software development in C.

Another consequence of being immersed within existing practice is that developers use the characteristics of different entities to form categories. This categorization provides a mechanism for people to make generalizations based on relatively small data sets. A developer working with C source code which has been written by other people will slowly build up a set of assumptions and expectations of the general characteristics of this code.

A study by Nisbett, Krantz, Jepson and Kunda^[325] illustrates peoples propensity to generalize, based on past experience. Subjects were given the following scenario. (Some were told that three samples of each object was encountered, while other subjects were told that 20 samples of each object was encountered.)

Imagine that you are an explorer who has landed on a little-known island in the Southeastern Pacific. You encounter several new animals, people, and objects. You observe the properties of your "samples" and you need to make guesses about how common these properties would be in other animals, people, or objects of the same type:

1. suppose you encounter a new bird, the shreeble. It is blue in color. What percent of all shreebles on the island do you expect to be blue?
2. suppose the shreeble you encounter is found to nest in a eucalyptus tree, a type of tree that is fairly common on this island. What percentage of all shreebles on the island do you expect to nest in a eucalyptus tree?

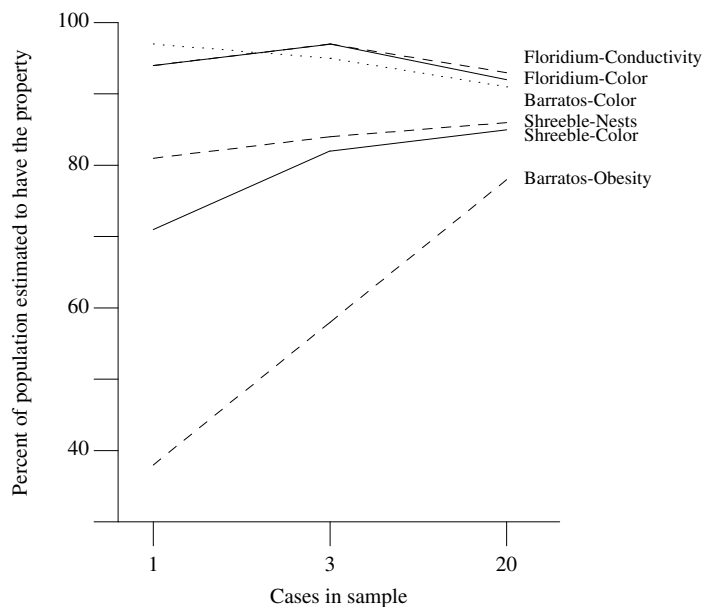


Figure 0.9: Percentage of population estimated to have the sample property against the number of cases in the sample. Adapted from Nisbett.^[325]

3. suppose you encounter a native, who is a member of a tribe called the Barratos. He is obese. What percentage of the male Barratos do you expect to be obese?
4. suppose the Barratos man is brown in color. What percentage of male Barratos do you expect be brown (as opposed to red, yellow, black, or white)?
5. suppose you encounter what the physicist on your expedition describes as an extremely rare element called floridium. Upon being heated to a very high temperature, it burns with a green flame. What percentage of all samples of floridium found on the island do you expect to burn with a green flame?
6. suppose the samples of floridium, when drawn into a filament, is found to conduct electricity. What percentage of all samples of floridium found on the island do you expect to conduct electricity?

The results show that subjects used their knowledge of the variability of properties in estimating the probability that an object would have that property. For instance, different samples of the same element are not expected to exhibit different properties, so the number of cases in a sample did not influence estimated probabilities. However, people are known to vary in their obesity, so the estimated probabilities were much lower for the single sample than the 20 case sample.

The lesson to be learned here is a general one concerning the functionality of objects and functions that are considered to form a category. Individual members of a category (e.g., a source file or structure type created by a developer) should have properties that would not be surprising to somebody who was only familiar with a subset of the members (see Figure 0.9).

Having expectations and making assumptions (or more technically, using inductive reasoning) can be useful in a slowly changing world (such as the one inhabited by our ancestors). They provide a framework from which small amounts of information can be used to infer, seemingly unconnected (to an outsider), conclusions. Is there a place for implicit expectations and assumptions in software development? A strong

case can be made for saying that any thought process that is not based on explicit knowledge (which can be stated) should not be used when writing software. In practice use of such knowledge, and inductive reasoning based on it, appears to play an integral role in human thought processes. A guideline recommendation that developers not use such thought processes may be difficult, if not impossible, to adhere to.

These coding guidelines don't seek to change what appears to be innate developer (human) behavior. The approach taken by these guidelines is to take account of the thought processes that developers use, and to work within them. If developers have expectations and make assumptions, then the way to deal with them is to find out what they are and to ensure that, where possible, source code follows them (or at least does not exhibit behavior that differs significantly from that expected). This approach means that these recommendations are tuned to the human way of comprehending C source code.

The issue of implicit knowledge occurs in several coding guidelines.

9.1.1 Aims and motivation

What are developers trying to do when they read and write source code? They are attempting to satisfy a variety of goals. These goals can be explicit or implicit. One contribution cognitive psychology can make is to uncover the implicit goals, and perhaps to provide a way of understanding their effects (with the aim of creating guideline recommendations that minimize any undesirable consequences). Possible developer aims and motives include (roughly from higher level to lower level) the following:

- Performing their role in a development project (with an eye on promotion, for the pleasure of doing a good job, or doing a job that pays for other interests).
- Carrying out a program-modification task.
- Extracting information from the source by explicitly choosing what to pay attention to.
- Minimizing cognitive effort; for instance, using heuristics rather than acquiring all the necessary information and using deductive logic.
- Maximizing the pleasure they get out of what they are doing.
- Belief maintenance: studies have found that people interpret evidence in ways that will maintain their existing beliefs.

The act of reading and writing software has an immediate personal cost. It is the cognitive load on a developer's brain (physical effort is assumed to be small enough that it has no significant cost, noticeable to the developer). Various studies have shown that people try to minimize cognitive effort when performing tasks.^[136] A possible consequence of minimizing this effort is that people's actions are not always those that would be predicted on the basis of correct completion of the task at hand. In other words, people make mistakes because they do not invest sufficient effort to carry out a task correctly.

When attempting to solve a problem, a person's cognitive system is assumed to make cost/accuracy trade-offs. The details of how it forms an estimate of the value, cost, and risk associated with an action, and carries out the trade-off analysis is not known. A study by Fu and Gray^[136] provides a good example of the effects of these trade-offs on the decisions made by people when performing a task. Subjects were given the task of copying a pattern of colored blocks (on a computer-generated display). To carry out the task subjects had to remember the color of the block to be copied and its position in the target pattern, a memory effort. A perceptual-motor effort was introduced by graying out the various areas of the display where the colored blocks were visible. These grayed out areas could be made temporarily visible using various combinations of keystrokes and mouse movements. When performing the task, subjects had the choice of expending memory effort (learning the locations of different colored blocks) or perceptual-motor effort (using keystrokes and mouse movements to uncover different areas of the display). A subject's total effort was equal to the sum of the perceptual motor effort and the memory storage and recall effort. The extremes of possible effort combinations are: (1) minimize the memory effort by remembering the color and position of a single block, which requires the perceptual-motor effort of uncovering the grayed out area for every block, or (2) minimize

statement
visual layout
declaration
visual layout
identifier
syntax
developer
motivations

cognitive
effort

belief mainte-
nance

cost/accuracy
trade-off

perceptual effort by remembering information on as many blocks as possible (this requires uncovering fewer grayed areas).

The subjects were split into three groups. The experiment was arranged such that one group had to expend a low effort to uncover the grayed out areas, the second acted as a control, and the third had to expend a high effort to uncover the grayed out areas. The results showed that the subjects who had to expend a high perceptual-motor effort, uncovered grayed out area fewer times than the other two groups. These subjects also spent longer looking at the areas uncovered, and moved more colored blocks between uncoverings. The subjects faced with a high perceptual-motor effort reduced their total effort by investing in memory effort. Another consequence of this switch of effort investment, to use of memory, was an increase in errors made.

When reading source code, developers may be faced with the same kind of decision. Having looked at and invested effort in memorizing information about a section of source code, should they invest perceptual-motor effort when looking at a different section of source that is affected by the previously read source to verify the correctness of the information in their memory? A commonly encountered question is the C language type of an object. A developer has to decide between searching for the declaration or relying on information in memory.

A study by Schunn, Reder, Nhouyvanisvong, Richards, and Stroffolino^[395] found that a subject's degree of familiarity with a problem was a better predictor, than retrievability of an answer, of whether subjects would attempt to retrieve or calculate the answer to a problem.

The issue of cognitive effort vs. accuracy in decision making is also discussed elsewhere.

Experience shows that many developers believe that code *efficiency* is an important attribute of code quality. This belief is not unique to the culture of C and has a long history.^[235] While efficiency remains an issue in some application domains, these coding guidelines often treat efficiency as a cause of undesirable developer behavior that needs to be considered (with a view handling the possible consequences).

Experience has shown that some developers equate visual compactness of source code with runtime efficiency of the translated program. While there are some languages where such a correlation exists (e.g., some implementations of Basic, mostly interpreter based and seen in early hobbyist computers, perform just in time translation of the source code), it does not exist for C. This is an issue that needs to be covered during developer education.

Experience has also shown that when presented with a choice developer decisions are affected by their own estimates of the amount of typing they will need to perform. Typing minimization behavior can include choosing abbreviated identifier names, using cut-and-paste to copy sections of code, using keyboard short-cuts, and creating editor macros (which can sometimes require significantly more effort than they save).

9.2 Selecting guideline recommendations

No attempt has been made to keep the number of guideline recommendations within a prescribed limit. It is not expected that developers should memorize them. Managers are expected to select guidelines based on their cost effectiveness for particular projects.

Leaving the number of guideline recommendations open-ended does not mean that any worthwhile sounding idea has been written up as a guideline. Although the number of different coding problems that could be encountered is infinite, an endless list of guidelines would be of little practical use. Worthwhile recommendations are those that minimize both the likelihood of faults being introduced by a developer or the effort needed by subsequent developers to comprehend the source code. Guideline recommendations covering situations that rarely occur in practice are wasted effort (not for the developers who rarely get to see them, but for the guideline author and tool vendors implementing checks for them).

These coding guidelines are not intended to recommend against the use of constructs that are obviously faults (i.e., developers have done something by mistake and would want to modify the code if the usage was pointed out to them). For instance, a guideline recommending against the use of uninitialized objects is equivalent to a guideline recommending against faults (i.e., pointless). Developers do not need to be given recommendations not to use these constructs. Guidelines either recommend against the use of constructs that are intentionally used (i.e., a developer did not use them by mistake) in a conforming program (any constructs

effort vs. accuracy
decision making

visually compact code
efficiency belief

typing minimization

guideline recommendations
selecting

guidelines not faults

that would cause a conforming translator to issue a diagnostic are not included), or they recommend that a particular implementation technique be used. diagnostic shall produce

These guidelines deal with the use of C language constructs, not the design decisions behind their selection. It is not the intent to discuss how developers choose to solve the higher-level design and algorithmic issues associated with software development. These guidelines deal with instances of particular constructs at the source code level.

Source code faults are nearly always clichés; that is, developers tend to repeat the mistakes of others and their own previous mistakes. Not every instance of a specific construct recommended against by a guideline (e.g., an assignment operator in a conditional expression, `if (x = y)`) need result in a fault. However, because a sufficient number of instances have caused faults to occur in the past, it is considered to be worthwhile recommending against all usage of a construct.

Guidelines covering a particular construct cannot be considered in isolation from the rest of the language. The question has to be asked, of each guideline: “if developers are not allowed do this, what are they going to do instead?” A guideline that effectively forces developers into using an even more dangerous construct is a lot more than simply a waste of time. For instance, your authors experience is that placing too many restrictions on how enumerated constants are defined leads to developers using macro names instead— a counterproductive outcome.

Selecting guideline recommendations based on the preceding criteria requires both a detailed inventory of software faults for the C language (no distinction is made between faults that are detected in the source and faults that are detected as incorrect output from a program) and some measure of developer comprehension effort. Developer comprehension is discussed elsewhere. There have been relatively few reliable studies of software faults (Knuth’s^[228] log of faults in $\text{T}_{\text{E}}\text{X}$ is one such; see Fredericks^[134] for a survey). Some of those that have been published have looked at faults that occur during initial development,^[452] and faults that occur during the evolution of an application, its maintenance period.^[161,346] 0 developer program comprehension

Guidelines that look worthy but lack empirical evidence for their cost effectiveness should be regarded with suspicion. The field of software engineering has a poor track record for experimental research. Studies^[269,505] have found that most published papers in software related disciplines do not include any experimental validation. Whenever possible this book quotes results based on empirical studies (for the measurements by the author, either the raw data or the source code of the programs that generated the data are available from the author^[506]). Sometimes results from theoretical derivations are used. As a last resort, common practices and experience are sometimes quoted. Those studies that have investigated issues relating to coding practices have often used very inexperienced subjects (students studying at a university). The results of these inexperienced subject-based studies have been ignored. 0 experimental studies

Table 0.2: Fault categories ordered by frequency of occurrence. The last column is the rank position after the fault fix weighting factor is taken into account. Based on Perry.^[346]

Rank	Fault Description	% Total Faults	Fix Rank	Rank	Fault Description	% Total Faults	Fix Rank
1	internal functionality	25.0	13	12	error handling	3.3	6
2	interface complexity	11.4	10	13	primitive’s misuse	2.4	11
3	unexpected dependencies	8.0	4	14	dynamic data use	2.1	15
4	low-level logic	7.9	17	15	resource allocation	1.5	2
5	design/code complexity	7.7	3	16	static data design	1.0	19
6	other	5.8	12	17	performance	0.9	1
7	change coordinates	4.9	14	18	unknown interactions	0.7	5
8	concurrent work	4.4	9	19	primitives unsupported	0.6	19
9	race conditions	4.3	7	20	IPC rule violated	0.4	16
10	external functionality	3.6	8	21	change management complexity	0.3	21
11	language pitfalls i.e., use of = when == intended	3.5	18	22	dynamic data design	0.3	21

Usage
1

- A study by Thayer, Lipow, and Nelson^[452] looked at several Jovial (a Fortran-like language) projects during their testing phase. It was advanced for its time, using tools to analyze the source and being rigorous in the methodology of its detailed measurements. The study broke new ground: “Based on error histories seen in the data, define sets of error categories, both causative and symptomatic, to be applied in the analysis of software problem reports and their closure.” Unfortunately, the quality of this work was not followed up by others and the level of detail provided is not sufficient for our needs here.
- Hatton^[161] provides an extensive list of faults in C source code found by a static analysis tool. The tool used was an earlier version of one of the tools used to gather the usage information for this book.
- Perry^[346] looked at the modification requests for a 1 MLOC system that contained approximately 15% to 20% new code for each release. As well as counting the number of occurrences of each fault category, a weight was given to the effort required to fix them.

Table 0.3: Underlying cause of faults. The *none given* category occurs because sometimes both the fault and the underlying cause are the same. For instance, *language pitfalls*, or *low-level logic*. Based on Perry.^[346]

Rank	Cause Description	% Total Causes	Fix Rank
1	Incomplete/omitted design	25.2	3
2	None given	20.5	10
3	Lack of knowledge	17.8	8
4	Ambiguous design	9.8	9
5	Earlier incorrect fix	7.3	7
6	Submitted under duress	6.8	6
7	Incomplete/omitted requirements	5.4	2
8	Other	4.1	4
9	Ambiguous requirements	2.0	1
10	Incorrect modifications	1.1	5

Looking at the results (shown in Table 0.2) we see that although performance is ranked 17th in terms of number of occurrences, it moves up to first when effort to fix is taken into account. Resource allocation also moves up the rankings. The application measured has to operate in realtime, so performance and resource usage will be very important. The extent to which the rankings used in this case apply to other application domains is likely to depend on the application domain. Perry also measured the underlying causes (see Table 0.3) and the means of fault prevention (see Table 0.4).

Table 0.4: Means of fault prevention. The last column is the rank position after the fault fix weighting factor is taken into account. Based on Perry.^[346]

Rank	Means Description	% Observed	Fix Rank
1	Application walk-through	24.5	8
2	Provide expert/clearer documentation	15.7	3
3	Guideline enforcement	13.3	10
4	Requirements/design templates	10.0	5
5	Better test planning	9.9	9
6	Formal requirements	8.8	2
7	Formal interface specifications	7.2	4
8	Other	6.9	6
9	Training	2.2	1
10	Keep document/code in sync	1.5	7

- A study by Glass^[145] looked at what he called *persistent software errors*. Glass appears to make an implicit assumption that faults appearing late in development or during operational use are somehow

different from those found during development. The data came from analyzing *software problem reports* from two large projects. There was no analysis of faults found in these projects during development.

Your author knows of no study comparing differences in faults found during early development, different phases of testing and operational use. Until proven otherwise, these Coding guideline subsections treat the faults found during different phases of development as having the same characteristics.

More detailed information on the usage of particular C constructs is given in the Usage sections of this book. While this information provides an estimate of the frequency-of-occurrence of these constructs, it does not provide any information on their correlation to occurrences of faults. These frequency of occurrence measurements were used in the decision process for deciding when particular constructs might warrant a guideline (the extent to which frequency of occurrence might affect developer performance. Note that power law of learning is not considered here.

The selection of these guidelines was also influenced by the intended audience of developers, the types of programs they work on, and the priorities of the environment in which these developers work as follows:

- Developers are assumed to have imperfect memory, work in a fashion that minimizes their cognitive load, are not experts in C language and are liable to have incorrect knowledge about what they think C constructs mean; and have an incomplete knowledge base of the sources they are working on. Although there may be developers who are experts in C language and the source code they are working on, it is assumed here that such people are sufficiently rare that they are not statistically significant; in general these Coding guideline subsections ignore them. A more detailed discussion is given elsewhere.
- Applications are assumed to be large (over 50 KLOC) and actively worked on by more than one developer.
- Getting the software right is only one of the priorities in any commercial development group. Costs and time scales need to be considered. Following coding guidelines is sometimes a small component of what can also be a small component in a big project.

9.2.1 Guideline recommendations must be enforceable

A guideline recommendation that cannot be enforced is unlikely to be of any use. Enforcement introduces several practical issues that constrain the recommendations made by guidelines, including the following:

- *Detecting violations.* It needs to be possible to deduce (by analyzing source code) whether a guideline is, or is not, being adhered to. The answer should always be the same no matter who is asking the question (i.e., the guidelines should be unambiguous).
- *Removing violations.* There needs to be a way of rewriting the source so that no guideline is violated. Creating a situation where it is not possible to write a program without violating one or other guidelines debases the importance of adhering to guidelines and creates a work environment that encourages the use of deviations.
- *Testing modified programs.* Testing can be a very expensive process. The method chosen, by developers, to implement changes to the source may be based on minimizing the possible impact on other parts of a program, the idea being to reduce the amount of testing that needs to be done (or at least that appears to be needed to be done). Adhering to a guideline should involve an amount of effort that is proportional to the effort used to make changes to the source. Guidelines that could require a major source restructuring effort, after a small change to the source, are unlikely to be adhered to.

The procedures that might be followed in checking conformance to guidelines are not discussed in this book. A number of standards have been published dealing with this issue.^[184, 185, 193]

A project that uses more than a handful of guidelines will find it uneconomical and impractical to enforce them without some form of automated assistance. Manually checking the source code against all guidelines is likely to be expensive and error prone (it could take a developer a working week simply to learn the guidelines, assuming 100 rules and 20 minutes study of each rule). Recognizing that some form of automated tool will be used, the wording for guidelines needs to be algorithmic in style.

There are situations where adhering to a guideline can get in the way of doing what needs to be done. Adhering to coding guidelines rarely has the highest priority in a commercial environment. Experience has shown that these situations can lead either to complete guideline recommendations being ignored, or be the thin end of the wedge that eventually leads to the abandonment of adherence to any coding guideline. The solution is to accept that guidelines do need to be broken at times. This fact should not be swept under the carpet, but codified into a deviation mechanism.

9.2.1.1 Uses of adherence to guidelines

While reducing the cost of ownership may be the aim of these guideline recommendations, others may see them as having other uses. For instance, from time to time there are calls for formal certification of source code to some coding guideline document or other. Such certification has an obvious commercial benefit to the certification body and any associated tools vendors. Whether such certification provides a worthwhile benefit to purchasers of software is debatable.^[486]

Goodhart's law^{0.1} deals with the impact of external, human pressure on measurement and is applicable here. One of its forms is: "When a measure becomes a target, it ceases to be a good measure." Strathern^[434] describes how the use of a rating system changed the nature of university research and teaching.

Whether there is a greater economic benefit, to a company, in simply doing what is necessary to gain some kind of external recognition of conformance to a coding guideline document (i.e., giving little weight to the internal cost/benefit analysis at the source code level), or in considering adherence to guideline recommendations as a purely internal cost/benefit issue is outside the scope of this book.

9.2.1.2 Deviations

A list of possible deviations should be an integral part of any coding guideline. This list is a continuation of the experience and calculation that forms part of every guideline.

The arguments made by the advocates of Total Quality Management^[263] appear to be hard to argue against. The relentless pursuit of quality is to be commended for some applications, such as airborne systems and medical instruments. Even in other, less life-threatening, applications, quality is often promoted as a significant factor in enhancing customer satisfaction. Who doesn't want fault-free software? However, in these quality discussions, the most important factor is often overlooked— financial and competitive performance— (getting a product to market early, even if it contains known faults, is often much more important than getting a fault-free product to market later). Delivering a fault-free product to market late can result in financial ruin, just as delivering a fault prone product early to market. These coding guidelines aim of reducing the cost of software ownership needs to be weighed against the broader aim of creating value in a timely fashion. For instance, the cost of following a particular guideline may be much greater than normal, or an alternative technique may not be available. In these situations a strong case can be made for not adhering to an applicable guideline.

There is another practical reason for listing deviations. Experience shows that once a particular guideline has not been adhered to in one situation, developers find it easier not to adhere to it in other situations. Management rarely has access to anybody with sufficient expertise to frame a modified guideline (deviation) appropriate to the situation, even if that route is contemplated. Experience shows that developers rarely create a subset of an individual guideline to ignore; the entire guideline tends to be ignored. A deviation can stop adherence to a particular guideline being an all-or-nothing decision, helping to prevent the leakage of

^{0.1}Professor Charles Goodhart, FBA, was chief adviser to the Bank of England and his "law" was originally aimed at financial measures (i.e., "As soon as the government attempts to regulate any particular set of financial assets, these become unreliable as indicators of economic trends.").

nonadherence. Deviations can provide an incremental sequence (increasing in cost and benefit) of decision points.

Who should decide when a deviation can be used? Both the authors of the source code and their immediate managers may have a potential conflict of interest with the longer-term goals of those paying for the development as follows:

- They may be under pressure to deliver a release and see use of a deviation as a short-cut.
- They may not be the direct beneficiaries of the investment being made in adhering to coding guidelines. Redirecting their resources to other areas of the project may seem attractive.
- They may not have the skill or resources needed to follow a guideline in a particular case. Admitting one's own limitations is always hard to do.

The processes that customers (which may be other departments within the same company) put in place to ensure that project managers and developers follow agreed-on practices are outside the scope of this book. Methods for processing deviation requests include:

- referring all requests to an expert. This raises the question of how qualified a *C expert* must be to make technical decisions on deviations.
- making deviation decisions during code review.
- allowing the Q/A department to have the final say about which deviations are acceptable.

o developer
expertise

However, permission for the use of a deviation is obtained, all uses need to be documented. That is, each source construct that does not adhere to the full guideline, but a deviation of that guideline, needs to be documented. This documentation may simply be a reference to one location where the rationale for that deviation is given. Creating this documentation offers several benefits:

- It ensures that a minimum amount of thought has been given to the reasons for use of a deviation.
- It may provide useful information to subsequent developers. For instance, it can provide an indication of the number of issues that may need to be looked at when porting to a new translator, and the rationale given with a deviation can provide background information on coding decisions.
- It provides feedback to management on the practical implications of the guidelines in force. For instance, is more developer training required and/or should particular guidelines be reviewed (and perhaps reworded)?

Information given in the documentation for a deviation may need to include the following:

- The cost/benefit of following the deviation rather than the full guideline, including cost estimates.
- The risks associated with using the deviation rather than the full guideline recommendation.
- The alternative source code constructs and guidelines considered before selecting the deviation.

9.2.2 Code reviews

Some coding guidelines are not readily amenable to automatic enforcement. This can occur either because they involve trade-offs among choices, or because commercial tool technology is not yet sufficiently advanced. The solution adopted here is to structure those guidelines that are not amenable to automatic enforcement so that they can be integrated into a code review process.

code reviews

It is expected that those guideline recommendation capable of being automatically checked will have been enforced before the code is reviewed. Looking at the output of static analysis tools during code review is

usually an inefficient use of human resources. It makes sense for the developers writing the source code to use the analysis tools regularly, not just prior to reviews.

These coding guidelines are not intended to cover all the issues that should be covered during reviews. Problems with the specification, choice of algorithms, trade-offs in using constructs, agreement with the specification, are among the other issues that should be considered.

justifying
decisions

The impact of code reviews goes beyond the immediate consequences of having developers read and comment on each other's code. Knowing that their code is to be reviewed by others can affect developer's decision—making strategy. Even hypothetical questions raised during a code review can change subsequent decision making.^[129]

Code reviews are subject to the same commercial influences as other development activities; they require an investment of resources (a cost) to deliver benefits. Code reviews are widely seen as a good idea and are performed by many development groups. A very common rationale given for having code reviews is that they are a cost effective means of detecting faults. A recent review^[354] questioned this assumption, based on the lack of experimental evidence showing it to be true. Another reason for performing code reviews is the opportunity it provides for junior developers learn the culture of a development group.

Organizations that have a formal review procedure often follow a three-stage process of preparation, collection, and repair. During preparation, members of the review team read the source looking for as many defects as possible. During review the team as a whole looks for additional defects and collates a list of agreed-on defects. Repair is the resolution of these defects by the author of the source.

Studies by Porter, Siy, Mockuss, and Votta^[355–357] to determine the best form for code reviews found that: inspection interval and effectiveness of defect detection were not significantly affected by team size (large vs. small), inspection interval and effectiveness of defect detection were not significantly affected by the number of sessions (single vs. multiple), and the effectiveness of defect detection was not improved by performing repairs between sessions of two-session inspections (however, inspection interval was significantly increased). They concluded that single-session inspections by small teams were the most efficient because their defect-detection rate was as good as other formats, and inspection interval was the same or less.

9.3 Relationship among guidelines

coding guidelines
relationship
among

Individual guideline recommendations do not exist in isolation. They are collected together to form a set of coding guidelines. Several properties are important in a set of guideline recommendations, including:

- It must be possible to implement the algorithmic functionality required by one guideline without violating any of the guidelines in a set.
- Consistency among guidelines within a set is a worthwhile aim.
- Being able to use the same process to enforce all requirements within a set of guidelines is a worthwhile aim.

As a complete set, the guideline recommendations in this book do not meet all of these requirements, but it is possible to create a number of sets that do meet them. It is management's responsibility to select the subset of guidelines applicable to their development situation.

9.4 How do guideline recommendations work?

guideline rec-
ommendations
how they work

How can adhering to these coding guidelines help reduce the cost of software ownership? The following are possible mechanisms:

- Reduce the number of faults introduced into source code by recommending against the use of constructs known to have often been the cause of faults in the past. For instance, by recommending against the use of an assignment operator in a conditional expression, `if (x = y)`.
- Developers have different skills and backgrounds. Adhering to guidelines does not make developers write good code, but these recommendations can help prevent them from writing code that will be more costly than necessary to maintain.

- Developers' programming experience is often limited, so they do not always appreciate all the implications of using constructs. Guideline recommendations provide a prebuilt knowledge net. For instance, they highlight constructs whose behavior is not as immutable as developers might have assumed. The most common response your author hears from developers is "Oh, I didn't know that".

The primary purpose of coding guidelines is not usually about helping the original author of the code (although as a user of that code they can be of benefit to that person). Significantly more time and effort are spent maintaining existing programs than in writing new ones. For code maintenance, being able to easily extract information from source code, in order to predict the behavior of a program (sometimes called *program comprehension*), is an important issue.

Does reducing the cognitive effort needed to comprehend source code increase the rate at which developers comprehend it and/or reduce the number of faults they introduce into it? While there is no direct evidence proving that it does, these coding guideline subsections assume that it does.

9.5 Developer differences

To what extent do individual developer differences affect the selection and wording of coding guidelines? To answer this question some of the things we would need to know include the following:

developer
differences

- the attributes that vary between developers,
- the number of developers (ideally the statistical distribution) having these different attributes and to what extent they possess them, and
- the affect these attribute differences have on developers' performance when working with source code.

Psychologists have been studying and measuring various human attributes for many years. These studies are slowly leading to a general understanding of how human cognitive processes operate. Unfortunately, there is no experimentally verified theory about the cognitive processes involved in software development. So while a lot of information on the extent of the variation in human attributes may be known, how these differences affect developers' performance when working with source code is unknown.

The overview of various cognitive psychology studies, appearing later in this introduction, is not primarily intended to deal with differences between developers. It is intended to provide a general description of the characteristics of the mental processing capabilities of the human mind. Strengths, weaknesses, and biases in these capabilities need to be addressed by guidelines. Sometimes the extent of individuals' capabilities do vary significantly in some areas. Should guidelines address the lowest common denominator (anybody could be hired), or should they assume a minimum level of capability (job applicants need to be tested to ensure they are above this level)?

What are the costs involved in recommending that the capabilities required to comprehend source code not exceed some maximum value? Do these costs exceed the likely benefits? At the moment these questions are somewhat hypothetical. There are no reliable means of measuring developers' different capabilities, as they relate to software development, and the impact of these capabilities on the economics of software development is very poorly understood. Although the guideline recommendations do take account of the capability limitations of developers, they are frustratingly nonspecific in setting boundaries.

These guidelines assume some minimum level of knowledge and programming competence on the part of developers. They do not require any degree of expertise (the issue of expertise is discussed elsewhere).

o developer
expertise

- A study by Monaghan^[307,308] looked at measures for discriminating *ability* and *style* that are relevant to representational and strategy differences in people's problem solving.
- A study by Oberlander, Cox, Monaghan, Stenning, and Tobin^[329] investigated student responses to multimodal (more than one method of expression, graphical and sentences here) logic teaching. They found that students' preexisting cognitive styles affected both the teaching outcome and the structure of their logical discourse.

- A study by MacLeod, Hunt and Mathews^[273] looked at sentence–picture comprehension. They found one group of subjects used a comprehension strategy that fit a linguistic model, while another group used a strategy that fit a pictorial–spatial model. A psychometric test of subjects showed a high correlation between the model a subject used and their spatial ability (but not their verbal ability). Sentence–picture comprehension is discussed in more detail elsewhere. In most cases C source visually appears, to readers, in a single mode, linear text. Although some tools are capable of displaying alternative representations of the source, they are not in widespread use. The extent to which a developer’s primary mode of thinking may affect source code comprehension in this form is unknown.

The effect of different developer personalities is discussed elsewhere, as are working memory, reading span, rate of information processing, the affects of age, and cultural differences. Although most developers are male,^[92] gender differences are not discussed.

9.6 What do these guidelines apply to?

A program (at least those addressed by these Coding guidelines) is likely to be built from many source files. Each source file is passed through eight phases of translation. Do all guidelines apply to every source file during every phase of translation? No, they do not. Guideline recommendations are created for a variety of different reasons and the rationale for the recommendation may only be applicable in certain cases; for instance:

- Reduce the cognitive effort needed to comprehend a program usually apply to the visible source code. That is, the source code as viewed by a reader, for example, in an editor. The result of preprocessing may be a more complicated expression, or sequence of nested constructs than specified by a guideline recommendation. But, because developers are not expected to have to read the output of the preprocessor, any complexity here may not be relevant,
- Common developer mistakes may apply during any phase of translation. The contexts should be apparent from the wording of the guideline and the construct addressed.
- Possible changes in implementation behavior can apply during any phase of translation. The contexts should be apparent from the wording of the guideline and the construct addressed.
- During preprocessing, the sequence of tokens output by the preprocessor can be significantly different from the sequence of tokens (effectively the visible source) input into it. Some guideline recommendations apply to the visible source, some apply to the sequence of tokens processed during syntax and semantic analysis, and some apply during other phases of translation.
- Different source files may be the responsibility of different development groups. As such, they may be subject to different commercial requirements, which can affect management’s choice of guidelines applied to them.
- The contents of system headers are considered to be opaque and outside the jurisdiction of these guideline recommendations. They are provided as part of the implementation and the standard gives implementations the freedom to put more or less what they like into them (they could even contain some form of precompiled tokens, not source code). Developers are not expected to modify system headers.
- Macros defined by an implementation (e.g., specified by the standard). The sequence of tokens these macros expand to is considered to be opaque and outside the jurisdiction of these coding guidelines. These macros could be defined in system headers (discussed previously) or internally within the translator. They are provided by the implementation and could expand to all manner of implementation-defined extensions, unspecified, or undefined behaviors. Because they are provided by an implementation, the intended actual behavior is known, and the implementation supports it. Developers can use these macros at the level of functionality specified by the standard and not concern themselves with implementation details.

sentence-
picture re-
lationships

developer
personality
memory
developer
reading span
developer
computational
power
identifier
information
extraction
memory
ageing
reason-
ing ability
age-related
catego-
rization
cultural differences
coding guidelines
what applied to?
source files
translation
phases of

prepro-
cessingheader
precompiled

Applying these reasons in the analysis of source code is something that both automated guideline enforcement tools and code reviewers need to concern themselves with.

It is possible that different sets of guideline recommendations will need to be applied to different source files. The reasons for this include the following:

- The cost effectiveness of particular recommendations may change during the code's lifetime. During initial development, the potential savings may be large. Nearer the end of the application's useful life, the savings achieved from implementing some recommendations may no longer be cost effective.
- The cost effectiveness of particular coding guidelines may vary between source files. Source containing functions used by many different programs (e.g., application library functions) may need to have a higher degree of portability, or source interfacing to hardware may need to make use of representation information.
- The source may have been written before the introduction of these coding guidelines. It may not be cost effective to modify the existing source to adhere to all the guidelines that apply to newly written code.

It is management's responsibility to make decisions regarding the cost effectiveness of applying the different guidelines under differing circumstances.

Some applications contain automatically generated source code. Should these coding guidelines apply to this kind of source code? The answer depends on how the generated source is subsequently used. If it is treated as an invisible implementation detail (i.e., the fact that C is generated is irrelevant), then C guideline recommendations do not apply (any more than assembler guidelines apply to C translators that chose to generate assembler as an intermediate step on the way to object code). If the generated source is to be worked on by developers, just like human-written code, then the same guidelines should be applied to it as to human written code.

9.7 When to enforce the guidelines

Enforcing guideline recommendations as soon as possible (i.e., while developers are writing the code) has several advantages, including:

coding guidelines
when to enforce

- Providing rapid feedback has been shown^[171] to play an essential role in effective learning. Having developers check their own source provides a mechanism for them to obtain this kind of rapid feedback.
- Once code-related decisions have been made, the cost of changing them increases as time goes by and other developers start to make use of them.
- Developers' acceptance is increased if their mistakes are not made public (i.e., they perform the checking on their own code as it is written).

It is developers' responsibility to decide whether to check any modified source before using the compiler, or only after a large number of modifications, or at some other decision point. Checking in source to a version-control system is the point at which its adherence to guidelines stops being a private affair.

To be cost effective, the process of checking source code adherence to guideline recommendations needs to be automated. However, the state of the art in static analysis tools has yet to reach the level of sophistication of an experienced developer. Code reviews are the suggested mechanism for checking adherence to some recommendations. An attempt has been made to separate out those recommendations that are probably best checked during code review. This is not to say that these guideline recommendations should not be automated, only that your author does not think it is practical with current, and near future, static analysis technology.

The extent to which guidelines are automatically enforceable, using a tool, depends on the sophistication of the analysis performed; for instance, in the following (use of uninitialized objects is not listed as a guideline recommendation, but it makes for a simple example):

```

1  extern int glob;
2  extern int g(void);
3
4  void f(void)
5  {
6  int loc;
7
8  if (glob == 3)
9    loc = 4;
10 if (glob == 3)
11    loc++;          /* Does loc have a defined value here? */
12 if (glob == 4)
13    loc--;          /* Does loc have a defined value here? */
14 if (g() == 2)
15    loc = 9;
16 if (g() == glob)
17    ++loc;
18 }
```

The existing value of `loc` is modified when certain conditions are true. Knowing that it has a defined value requires analysis of the conditions under which the operations are performed. A static analysis tool might: (1) mark objects having been assigned to and have no knowledge of the conditions involved; (2) mark objects as assigned to when particular conditions hold, based on information available within the function that contains their definition; (3) the same as (2) but based on information available from the complete program.

9.8 Other coding guidelines documents

coding guidelines
other documents

The writing of coding guideline documents is a remarkably common activity. Publicly available documents discussing C include,^[131, 161, 176, 198, 219, 230, 284, 299, 300, 350, 351, 369, 371, 373, 425, 433] and there are significantly more documents internally available within companies. Such guideline documents are seen as being a *good thing* to have. Unfortunately, few organizations invest the effort needed to write technically meaningful or cost-effective guidelines, they then fail to make any investment in enforcing them.^{0.2}

The following are some of the creators of coding guideline include:

- *Software development companies.*^[407] Your author's experience with guideline documents written by development companies is that at best they contain well-meaning platitudes and at worse consist of a hodge-podge of narrow observations based on their authors' experiences with another language.
- *Organizations, user groups and consortia that are users of software.*^[358, 497] Here the aim is usually to reduce costs for the organization, not software development companies. Coding guidelines are rarely covered in any significant detail and the material usually forms a chapter of a much larger document. Herrmann^[167] provides a good review of the approaches to software safety and reliability promoted by the transportation, aerospace, defense, nuclear power, and biomedical industries through their published guidelines.
- *National and international standards.*^[195] Perceived authority is an important attribute of any guidelines document. Several user groups and consortia are actively involved in trying to have their documents adopted by national, if not international, standards bodies. The effort and very broad spectrum of consensus needed for publication as an International Standard means that documents are likely to be first adopted as National Standards.

The authors of some coding guideline documents see them as a way of making developers write good programs (whatever they are). Your author takes the view that adherence to guidelines can only help prevent mistakes being made and reduce subsequent costs.

^{0.2}If your author is told about the existence of coding guidelines while visiting a company's site, he always asks to see a copy; the difficulty his hosts usually have in tracking down a copy is testament to the degree to which they are followed.

Most guideline recommendations specify subsets, not supersets, of the language they apply to. The term *safe subset* is sometimes used. Perhaps this approach is motivated by the idea that a language already has all the constructs it needs, the desire not to invent another language, or simply an unwillingness to invest in the tools that would be needed to handle additional constructs (e.g., adding strong typing to a weakly typed language). The guidelines in this book have been written as part of a commentary on the C Standard. As such, they restrict themselves to constructs in that document and do not discuss recommendations that involve extensions.

Experience with more strongly typed languages suggests that strong typing does detect some kinds of faults before program execution. Although experimental tool support for stronger type checking of C source is starting to appear,^[267,319,398] little experience in its use is available for study. This book does not specify any guideline recommendations that require stronger type checking than that supported by the C Standard.

Quite a few coding guideline documents have been written for C++.^[82,164,242,292–294,301,335,352,438] It is interesting to note that these coding guideline documents concentrate almost exclusively on the object-oriented features of C++ (i.e., primarily those constructs not available in C). It is almost as if their authors believe that developers using C++ will not make any of the mistakes that C developers make, despite one language almost being a superset of the other.

Coding guideline documents for other languages include those for Ada,^[80,195] Cobol,^[324] Fortran,^[235] PERL,^[81] Prolog,^[86] and SQL.^[127]

9.8.1 *Those that stand out from the crowd*

The aims and methods used to produce coding guidelines documents vary. Many early guideline documents concentrated on giving advice to developers about how to write efficient code.^[235] The availability of powerful processors, coupled with large quantities of source code, has changed the modern (since the 1980s) emphasis to one of maintainability rather than efficiency. When efficiency is an issue, the differences between processors and compilers makes it difficult to give general recommendations. Vendors' reference manuals sometimes provide useful background advice.^[9,179] The Object Defect Classification^[71] covers a wide variety of cases and has been shown to give repeatable results when used by different people.^[110]

9.8.1.1 *Bell Laboratories and the 5ESS*

Bell Laboratories undertook a root-cause analysis of faults in the software for their 5ESS Switching System.^[502] The following were found to be the top three causes of faults, and their top two subcomponents:

measurements
5ESS

1. Execution/oversight— 38%, which in turn was broken down into inadequate attention to details (75%) and inadequate consideration to all relevant issues (11%).
2. Resource/planning— 19%, which in turn was broken down into not enough engineer time (76%) and not enough internal support (4%).
3. Education/training— 15%, which in turn was broken down into area of technical responsibility (68%) and programming language usage (15%).

In an attempt to reduce the number of faults, a set of “Code Fault Prevention Guidelines” and a “Coding Fault Inspection Checklist” were written and hundreds of engineers were trained in their use. These guideline recommendations were derived from more than 600 faults found in a particular product. As such, they could be said to be tuned to that product (nothing was said about how different root causes might evolve over time).

Based on measurements of previous releases of the 5ESS software and engineering cost per house to implement the guidelines (plus other bug inject countermeasures), it was estimated that for an investment of US\$100 K, a saving of US\$7 M was made in product rework and testing.

One of the interesting aspects of programs is that they can contain errors in logic and yet continue to perform their designated function; that is, faults in the source do not always show up as a perceived fault by the user of a program. Static analysis of code provides an estimate of the number of potential faults, but not all of these will result in reported faults.

Why did the number of faults reported in the 5ESS software drop after the introduction of these guideline recommendations? Was it because previous root causes were a good measure of future root-cause faults?

The guideline recommendations created do not involve complex constructs that required a deep knowledge of C. They are essentially a list of mistakes made by developers who had incomplete knowledge of C. The recommendations could be looked on as C language knowledge tuned to the reduction of faults in a particular application program. The coding guideline authors took the approach that it is better to avoid a problem area than expect developers to have detailed knowledge of the C language (and know how to deal with problem areas).

In several places in the guideline document, it is pointed out that particular faults had costly consequences. Although evidence that adherence to a particular set of coding guidelines would have prevented a costly fault provides effective motivation for the use of those recommendations, this form of motivation (often seen in coding guideline documents) is counter-productive when applied to individual guideline recommendations. There is rarely any evidence to show that the reason for a particular coding error being more expensive than another one is anything other than random chance.

9.8.1.2 MISRA

MISRA

MISRA (Motor Industry Software Reliability Association, <http://www.misra.org.uk>) published a set of *Guidelines for the use of the C language in vehicle based software*.^[299,300] These guideline recommendations were produced by a committee of interested volunteers and have become popular in several domains outside the automobile industry. For the most part, they are based on the implementation-defined, undefined, and unspecified constructs listed in Annex G of the C90 Standard. The guidelines relating to issues outside this annex are not as well thought through (the technicalities of what is intended and the impact of following a guideline recommendation).

There are now 15 or more vendors who offer products that claim to enforce compliance to the MISRA guidelines. At the time of this writing these tools are not always consistent in their interpretation of the wording of the guidelines. Being based on volunteer effort, MISRA does not have the resources to produce a test suite or provide timely responses to questions concerning the interpretation of particular guidelines.

9.8.2 Ada

Ada^o
using

Although the original purpose of the Ada language was to reduce total software ownership costs, its rigorous type checking and handling of runtime errors subsequently made it, for many, the language of choice for development of high-integrity systems. An ISO Technical Report^[195] (a TR does not have the status of a standard) was produced to address this market.

The rationale given in many of the *Guidance* clauses of this TR is that of making it possible to perform static analysis by recommending against the use of constructs that make such analysis difficult or impossible to perform. Human factors are not explicitly mentioned, although this could be said to be the major issue in some of the constructs discussed. Various methods are described as not being cost effective. The TR gives the impression that what it proposes is cost effective, although no such claim is made explicitly.

ISO/IEC TR
15942:2000

... , it can be seen that there are four different reasons for needing or rejecting particular language features within this context:

1. Language rules to achieve predictability,
2. Language rules to allow modelling,
3. Language rules to facilitate testing,
4. Pragmatic considerations.

This TR also deals with the broader issues of verification techniques, code reviews, different forms of static analysis, testing, and compiler validation. It recognizes that developers have different experience levels and sometimes (e.g., clause 5.10.3) recommends that some constructs only be used by experienced developers (nothing is said about how experience might be measured).

9.9 Software inspections

Software inspections, technical reviews, program walk-throughs (whatever the name used), all involve people looking at source code with a view to improving it. Some of the guidelines in this book are specified for enforcement during code reviews, primarily because automated tools have not yet achieved the sophistication needed to handle the constructs described.

Software inspections are often touted as a cost-effective method of reducing the number of defects in programs. However, their cost effectiveness, compared to other methods, is starting to be questioned. For a survey of current methods and measurements, see;^[240] for a detailed handbook on the subject, see.^[135]

During inspections a significant amount of time is spent reading — reading requirements, design documents, and source code. The cost of, and likely mistakes made during, code reading are factors addressed by some guideline recommendations. The following are different ways of reading source code, as it might be applied during code reviews:

- *Ad hoc reading techniques.* This is a catch-all term for those cases, very common in commercial environments, where the software is simply given to developers. No support tools or guidance is given on how they should carry out the inspection, or what they should look for. This lack of support means that the results are dependent on the skill, knowledge, and experience of the people at the meeting.
- *Checklist reading.* As its name implies this reading technique compares source code constructs against a list of issues. These issues could be collated from faults that have occurred in the past, or published coding guidelines such as the ones appearing in this book. Readers are required to interpret applicability of items on the checklist against each source code construct. This approach has the advantage of giving the reader pointers on what to look for. One disadvantage is that it constrains the reader to look for certain kinds of problems only.
- *Scenario-based reading.* Like checklist reading, scenario-based reading provides custom guidance.^[298] However, as well as providing a list of questions, a scenario also provides a description on how to perform the review. Each scenario deals with the detection of the particular defects defined in the custom guidance. The effectiveness of scenario-based reading techniques depends on the quality of the scenarios.
- *Perspective-based reading.* This form of reading checks source code from the point of view of the customers, or consumers, of a document.^[35] The rationale for this approach is that an application has many different stakeholders, each with their own requirements. For instance, while everybody can agree that software quality is important, reaching agreement on what the attributes of quality are can be difficult (e.g., timely delivery, cost effective, correct, maintainable, testable). Scenarios are written, for each perspective, listing activities and questions to ask. Experimental results on the effectiveness of perspective-based reading of C source in a commercial environment are given by Laitenberger and Jean-Marc DeBaud.^[239]
- *Defect-based reading.* Here different people focus on different defect classes. A scenario, consisting of a set of questions to ask, is created for each defect class; for instance, invalid pointer dereferences might be a class. Questions to ask could include; Has the lifetime of the object pointed to terminated? Could a pointer have the null pointer value in this expression? Will the result of a pointer cast be correctly aligned?
- *Function-point reading.* One study^[241] that compared checklist and perspective-based reading of code, using professional developers in an industrial context, found that perspective-based reading had a lower cost per defect found.

This book does not recommend any particular reading technique. It is hoped that the guideline recommendations given here can be integrated into whatever method is chosen by an organization.

10 Applications

coding guidelines
applications

Several application issues can affect the kind of guideline recommendations that are considered to be applicable. These include the application domain, the economics behind the usage, and how applications evolve over time. These issues are discussed next.

initialization
syntax

The use of C as an intermediate language has led to support for constructs that simplify the job of translation from other languages. Some of these constructs are specified in the standard (e.g., a trailing comma in initializer lists), while others are provided as extensions (e.g., gcc's support for taking the address of labels and being able to specify the **register** storage class on objects declared with file scope, has influenced the decision made by some translator implementors, of other languages to generate C rather than machine code^[99]).

10.1 Impact of application domain

Usage
1

Does the application domain influence the characteristics of the source code? This question is important because frequency of occurrence of constructs in source is one criterion used in selecting guidelines. There are certainly noticeable differences in language usage between some domains; for instance:

- *Floating point.* Many applications make no use of any floating-point types, while some scientific and engineering applications make heavy use of this data type.
- *Large initializers.* Many applications do not initialize objects with long lists of values, while the device driver sources for the Linux kernel contain many long initializer lists.

There have been studies that looked at differences within different industries (e.g., banking, aerospace, chemical^[162]). It is not clear to what extent the applications measured were unique to those industries (e.g., some form of accounting applications will be common to all of them), or how representative the applications measured might be to specific industries as a whole.

Given the problems associated with obtaining source code for the myriad of different application domains, and the likely problems with separating out the effects of the domain from other influences, your author decided to ignore this whole issue. A consequence of this decision is that these guideline recommendations are a union of the possible issues that can occur across all application domains. Detailed knowledge of the differences would be needed to build a set of guidelines that would be applicable to each application domain. Managers working within a particular application domain may want to select guidelines applicable to that domain.

10.2 Application economics

Coding guidelines are applicable to applications of all sizes. However, there are economic issues associated with the visible cost of enforcing guideline recommendations. For instance, the cost of enforcement is not likely to be visible when writing new code (the incremental cost is hidden in the cost of writing the code). However, the visible cost of ensuring that a large body of existing, previously unchecked, code can be significant.

The cost/benefit of adhering to a particular guideline recommendation will be affected by the economic circumstances within which the developed application sits. These circumstances include

- short/long expected lifetime of the application,
- relative cost of updating customers,
- quantity of source code,
- acceptable probability of application failure (adherence may not affect this probability, but often plays well in any ensuing court case), and
- expected number of future changes/updates.

There are so many possible combinations that reliable estimates of the effects of these issues, on the applicability of particular guidelines, can only be made by those involved in managing the development projects (the COCOMO cost-estimation model uses 17 cost factors, 5 scale factors, a domain-specific factor, and a count of the lines of code in estimating the cost of developing an application). The only direct economic issues associated with guidelines, in this book, we discussed earlier and through the choice of applications measured.

0 COCOMO
0 development context
0 Usage
1

10.3 Software architecture

The term *architecture* is used in a variety of software development contexts.^{0.3} The analogy with buildings is often made, “firm foundations laying the base for . . .”. This building analogy suggests a sense of direction and stability. Some applications do have these characteristics (in particular many of those studied in early software engineering papers, which has led to the view that most applications are like this). Many large government and institutional applications have this form (these applications are also the source of the largest percentage of published application development research).

software architecture

To remind readers, the primary aim of these coding guidelines is to minimize the cost of software ownership. Does having a good architecture help achieve this aim? Is it possible to frame coding guidelines that can help in the creation of good architecture? What is a good architecture?

What constitutes good software architecture is still being hotly debated. Perhaps it is not possible to predict in advance what the best architecture for a given application is. However, experience shows that in practice the customer can rarely specify exactly what it is they want in advance, and applications close to what they require are obviously not close enough (or they would not be paying for a different one to be written). Creating a good architecture, for a given application, requires knowledge of the whole and designers who know how to put together the parts to make the whole. In practice applications are very likely to change frequently; it might be claimed that applications only stop changing when they stop being used. Experience has shown that it is almost impossible to predict the future direction of application changes.

The conclusion to be drawn, for these observations, is that there are reasons other than incompetence for applications not to have any coherent architecture (although at the level of individual source files and functions this need not apply). In a commercial environment, profitability is a much stronger motive than the desire for coherent software architecture.

Software architecture, in the sense of organizing components into recognizable structures, is relevant to reading and writing source in that developers’ minds also organize the information they hold. People do not store information in long-term memory as unconnected facts. These coding guidelines assume that having programs structured in a way that is compatible with how information is organized in developers’ minds, and having the associations between components of a program correspond to how developers make associations between items of information, will reduce the cognitive effort of reading source code. The only architectural and organizational issues considered important by the guideline recommendations in this book are those motivated by the characteristics of developers’ long-term memory storage and retrieval.

0 memory developer

0 categorization

For a discussion of the pragmatics of software architecture, see Foote.^[130]

10.3.1 Software evolution

Applications that continue to be used tend to be modified over time. The term *software evolution* is sometimes used to describe this process. Coding guidelines are intended to reduce the costs associated with modifying source. What lessons can be learned from existing applications that have evolved?

application evolution

There have been several studies that looked at the change histories of some very large (several million

^{0.3}Some developers like to refer to themselves as software architects. In the UK such usage is against the law, “. . . punishable by a fine not exceeding level 4 on the standard scale . . .” (Architects Act 1997, Part IV):

Use of title “architect”.

20. – (1) A person shall not practise or carry on business under any name, style or title containing the word “architect” unless he is a person registered under this Act.

(2) Subsection (1) does not prevent any use of the designation “naval architect”, “landscape architect” or “golf-course architect”.

line,^[137] or a hundred million^[106]) programs over many years,^[106,156,331] and significant growth over a few years.^[147] Some studies have simply looked at the types of changes and their frequency. Others have tried to correlate faults with the changes made. None have investigated the effect of source characteristics on the effort needed to make the changes.

The one thing that is obvious from the data published to date: Researchers are still in the early stages of working out which factors are associated with software evolution.

- A study^[95] at Bell Labs showed the efficiency gains that could be achieved using developers who had experience with previous releases over developers new to a project. The results indicated that developers who had worked on previous releases spent 20% of their time in project discovery work. This 20% was put down as the cost of working on software that was evolving (the costs were much higher for developers not familiar with the project).
- Another Bell Labs study^[305] looked at predicting the risk of introducing a fault into an existing software system while performing an update on it. They found that the main predictors were the number of source lines affected, developer experience, time needed to make the change, and an attribute they called *diffusion*. Diffusion was calculated from the number of subsystems, modules, and files modified during the change, plus the number of developers involved in the work. Graves^[150] also tried to predict faults in an evolving application. He found that the fault potential of a module correlated with a weighted sum of the contributions from all the times the module had been changed (recent changes having the most weight). Similar findings were obtained by Ohlsson.^[330,331]
- Lehman has written a number of papers^[253] on what he calls the *laws of software evolution*. Although they sound plausible, these “laws” are based on empirical findings from relatively few projects.
- Kemerer and Slaughter^[218] briefly review existing empirical studies and also describe the analysis of 25,000 change events in 23 commercial software systems (Cobol-based) over a 20-year period.
- Other studies have looked at the interaction of module coupling and cohesion with product evolution.

11 Developers

The remainder of this coding guidelines subsection has two parts. This first major subsection discusses the tasks that developers perform, the second (the following major subsection) is a review of psychology studies carried out in human characteristics of relevance to reading and writing source code. There is an academic research field that goes under the general title *the psychology of programming*; few of the research results from this field have been used in this book for reasons explained elsewhere. However, without being able to make use of existing research applicable to commercial software development, your author has been forced into taking this two-part approach; which is far from ideal. A consequence of this approach is that it is not possible to point at direct experimental evidence for some of the recommendations made in coding guidelines. The most that can be claimed is that there is a possible causal link between specific research results, cognitive theories, and some software development activities.

Although these coding guidelines are aimed at a particular domain of software development, there is no orientation toward developers having any particular kinds of mental attributes. It is hoped that this discussion will act as a stimulus for research aimed at the needs of commercial software development, which cannot take place unless commercial software developers are willing to give up some of their time to act as subjects (in studies). It is hoped that this book will persuade readers of the importance of volunteering to take part in this research.

11.1 What do developers do?

In this book, we are only interested in developer activities that involve source code. Most studies,^[344] the time spent on these activities does not usually rise above 25%, of the total amount of time developers spend on all activities. The non-source code-related activities, the other 75%, are outside the scope of this book. In this

software de-
velopment
expertise

coupling and
cohesion

coding guidelines
developers

psychol-
ogy of pro-
gramming

Usage
1
developer
0
differences

developers
what do they
do?

book, the reason for reading source code is taken to be that developers want to comprehend program behavior sufficiently well to be able to make changes to it. Reading programs to learn about software development, or for pleasure, are not of interest here.

The source that is eventually modified may be a small subset of the source that has been read. Developers often spend a significant amount of their time working out what needs to be modified and the impact the changes will have on existing code.^[95]

The tools used by developers to help them search and comprehend source tend to be relatively unsophisticated.^[410] This general lack of tool usage needs to be taken into account in that some of the tasks performed in a *manual-comprehension* process will be different from those carried out in a tool-assisted process.

The following properties are taken to be important attributes of source code, because they affect developer cognitive effort and load:

- *Readable*. Source is both scanned, looking for some construct, and read in a booklike fashion. The symbols appearing in the visible source need to be arranged so that they can be easily seen, recognized, and processed.
- *Comprehensible*. Having read a sequence of symbols in the source, their meaning needs to be comprehended.
- *Memorable*. With applications that may consist of many thousands of line of source code (100 KLOC is common), having developers continually rereading what they have previously read because they have forgotten the information they learned is not cost effective. Cognitive psychology has yet to come up with a model of human memory that can be used to calculate the memorability of source code. One practical approach might be to measure developer performance in reconstructing the source of a translation unit (an idea initially proposed by Shneiderman,^[404] who proposed a 90–10 rule—a competent developer should be able to reconstruct functionally 90% of a translation unit after 10 minutes of study).
- *Unsurprising*. Developers have expectations. Meeting those expectations reduces the need to remember special cases, and it reduces the possibility of faults caused by developers making assumptions (not checking that their expectations are true).

For a discussion of the issues involved in collecting data on developers' activities and some findings, see Dewayne^[345] and Bradac.^[50]

11.1.1 Program understanding, not

One of the first tasks a developer has to do when given source code is figure out what it does (the word *understand* is often used by developers). What exactly does it mean to understanding a program? The word *understanding* can be interpreted in several different ways; it could imply

- knowing all there is to know about a program. Internally (the source code and data structures) and externally— its execution time behavior.
- knowing the external behavior of a program (or perhaps knowing the external behavior in a particular environment), but having a limited knowledge of the internal behavior.
- knowing the internal details, but having a limited knowledge of the external behavior.

The concept of *understanding a program* is often treated as being a yes/no affair. In practice, a developer will know more than nothing and less than everything about a program. Source code can be thought of as a web of knowledge. By reading the source, developers acquire beliefs about it; these beliefs are influenced by their existing beliefs. Existing beliefs (many might be considered to be knowledge rather than belief, by the person holding them) can involve a programming language (the one the source is written in), general computing algorithms, and the application domain.

0 cognitive effort
0 cognitive load
reading kinds of

0 developer training

developer program comprehension

0 belief maintenance

When reading a piece of source code for the first time, a developer does not start with an empty set of beliefs. Developers will have existing beliefs, which will affect the interpretation given to the source code read. Developers learn about a program, a continuous process without a well-defined ending. This learning process involves the creation of new beliefs and the modification of existing ones. Using a term (*understanding*) that implies a yes/no answer is not appropriate. Throughout this book, the term *comprehension* is used, not *understanding*.

Program comprehension is not an end in itself. The purpose of the investment in acquiring this knowledge (using the definition of knowledge as “belief plus complete conviction and conclusive justification”) is for the developer to be in a position to be able predict the behavior of a program sufficiently well to be able to change it. Program comprehension is not so much knowledge of the source code as the ability to predict the effects of the constructs it contains (developers do have knowledge of the source code; for instance, knowing which source file contains a declaration).

While this book does not directly get involved in theories of how people learn, program comprehension is a learning process. There are two main theories that attempt to explain learning. Empirical learning techniques look for similarities and differences between positive and negative examples of a concept. Explanation-based learning techniques operate by generalizing from a single example, proving that the example is an instance of the concept. The proof is constructed by an inference process, making use of a domain theory, a set of facts, and logical implications. In explanation-based learning, generalizations retain only those attributes of an example that are necessary to prove the example is an instance of the concept. Explanation-based learning is a general term for learning methods, such as knowledge compilation and chunking, that create new concepts that deductively follow from existing concepts. It has been argued that a complete model of concept learning must have both an empirical and an explanation-based component.

What strategies do developers use when trying to build beliefs about (comprehend) a program? The theories that have been proposed can be broadly grouped into the following:

- *The top-down approach.* The developer gaining a top-level understanding of what the program does. Once this is understood, the developer moves down a level to try to understanding the components that implement the top level. This process is repeated for every component at each level until the lowest level is reached. A developer might chose to perform a depth-first or width-first analysis of components.
- *The bottom-up approach.* This starts with small sequences of statements that build a description of what they do. These descriptions are fitted together to form higher-level descriptions, and so on, until a complete description of the program has been built.
- *The opportunistic processors approach.* Here developers use both strategies, depending on which best suits the purpose of what they are trying to achieve.^[258]

There have been a few empirical studies, using experienced (in the industrial sense) subjects, of how developers comprehend code (the purely theoretically based models are not discussed here). Including:

- A study by Letovsky^[257] asked developers to talk aloud (their thoughts) as they went about the task of adding a new feature to a program. He views developers as *knowledge base understanders* and builds a much more thorough model than the one presented here.
- A study by Littman, Pinto, Letovsky and Soloway^[265] found two strategies in use by the developers (minimum of five years experience) they observed: In a systematic strategy the developers seek to obtain information about how the program behaves before modifying it; and in an as-needed strategy developers tried to minimize the effort needed to study the program to be modified by attempting to localize those parts of a program where the changes needed to be made. Littman et al. found that those developers using the systematic strategy outperformed those using the as-needed strategy for the 250-line program used in the experiment. They also noted the problems associated with attempting to use the systematic strategy with much larger programs.

- A study by Pennington^[342] investigated the differences in comprehension strategies used by developers who achieved high and low levels of program comprehension. Those achieving high levels of comprehension tended to think about both the application domain and the program (source code) domain rather than just the program domain. Pennington^[343] also studied mental representations of programs; for small programs she found that professional programmers built models based on control flow rather than data flow.
- A study by von Mayrhauser and Vans^[483,484] looked at experienced developers maintaining large, 40,000+ LOC applications and proposed an integrated code comprehension model. This model contained four major components, (1) program model, (2) situated model, (3) top-down model, and (4) knowledge base.
- A study by Shaft and Vessey^[397] gave professional programmer subjects source code from two different application domains (accounting and hydrology). The subjects were familiar with one of the domains but not the other. Some of the subjects used a different comprehension strategy for the different domains.

11.1.1.1 Comprehension as relevance

Programming languages differ from human languages in that they are generally viewed, by developers, as a means of one-way communication with a computer. Human languages have evolved for interactive communication between two, or more, people who share common ground.^{0.4}

relevance

One of the reasons why developers sometimes find source code comprehension so difficult is that the original authors did not write it in terms of a communication with another person. Consequently, many of the implicit assumptions present in human communication may not be present in source code. Relevance is a primary example. Sperber and Wilson^[424] list the following principles of human communication:

Principle of relevance

1. *Every act of ostensive communication communicates a presumption of its own optimal relevance.*

Sperber and Wilson^[424]

Presumption of optimal relevance

1. *The set of assumptions **I** which the communicator intends to make manifest to the addressee is relevant enough to make it worth the addressee's while to process the ostensive stimulus.*
2. *The ostensive stimulus is the most relevant one the communicator could have used to communicate **I**.*

A computer simply executes the sequence of instructions contained in a program image. It has no conception of application assumptions and relevance. The developer knows this and realizes that including such information in the code is not necessary. A common mistake made by novice developers is to assume that the computer is aware of their intent and will perform the appropriate operations. Teaching developers to write code such that can be comprehended by two very different addressee's is outside the scope of these coding guidelines.

program image

Source code contains lots of details that are relevant to the computer, but often of little relevance to a developer reading it. Patterns in source code can be used as indicators of relevance; recognizing these patterns is something that developers learn with experience. These coding guidelines do not discuss the teaching of such recognition.

Developers often talk of the *intended meaning* of source code, i.e., the meaning that the original author of the code intended to convey. Code comprehension being an exercise in obtaining an intended meaning that is assumed to exist. However, the only warranted assumption that can be made about source code is that the operations specified in it contribute to a meaning.

^{0.4}The study of meaning and communication between people often starts with Grice's maxims,^[151] but readers might find Sperber and Wilson^[424] easier going.

11.1.2 The act of writing software

The model of developers sitting down to design and then write software on paper, iterating through several versions before deciding their work is correct, then typing it into a computer is still talked about today. This method of working may have been necessary in the past because access to computer terminals was often limited and developers used paper implementations as a method of optimizing the resources available to them (time with, and without, access to a computer).

Much modern software writing is done sitting at a terminal, within an editor. Often no written, paper, notes are used. Everything exists either in the developer's head or on the screen in front of him (or her). However, it is not the intent of this book to suggest alternative working practices. Changing a system that panders to people's needs for short-term gratification,^[133] to one that delays gratification and requires more intensive periods of a difficult, painful activity (thinking) is well beyond your author's capabilities.

Adhering to guideline recommendation does not guarantee that high quality software will be written; it can only help reduce the cost of ownership of the software that is written.

ROI o These coding guidelines assume that the cost of writing software is significantly less than the cost of developer activities that occur later (testing, rereading, and modification by other developers). Adhering to guideline may increase the cost of writing software. The purpose of this investment is to make savings (which are greater than the costs by an amount proportional to the risk of the investment) in the cost of these later activities.

It is hoped that developers will become sufficiently fluent in using these guideline recommendations and that they will be followed automatically while entering code. A skilled developer should aim to be able to automatically perform as much of the code-writing process as possible. Performing these tasks automatically frees up cognitive resources for use on other problems associated with code development.

Alfred North Whitehead (1861–1947)

It is a profoundly erroneous truism . . . that we should cultivate the habit of thinking of what we are doing. The precise opposite is the case. Civilization advances by extending the number of important operations which we can perform without thinking about them.

developer o
flow

It is not suggested that the entire software development process take place without any thinking. The process of writing code can be compared to writing in longhand. The writer thinks of a sentence and his hand automatically writes the words. It is only schoolchildren who need to concentrate on the actual process of writing the words.

11.2 Productivity

productivity
developer

Although much talked about, there has been little research on individual developer productivity. There is the often quoted figure of a 25-to-1 productivity difference between developers; however, this is a misinterpretation of figures presented in two tables of a particular paper.^[149] Hopefully the analysis by Prechelt^[362] will finally put a stop to researchers quoting this large, incorrect, figure. The differences in performance found by Prechelt are rarely larger than four, similar to the performance ranges found by the original research.

Few measurement programs based on individual developers have been undertaken; many measures are based on complete projects, dividing some quantity (often lines of code) by the number of individuals working on them. See Scacchi^[391] for a review of the empirical software productivity research and Jones^[202] provides a good discussion of productivity over the complete life cycle of a project. However, some of the issues discussed (e.g., response time when editing source) are rooted in a mainframe environment and are no longer relevant.

Are there any guideline recommendations that the more productive developers use that we can all learn from? Your author knows of no published research that investigates productivity at this level of detail. Age-related productivity issues^[272,417] are not discussed in these coding guidelines. The subject of expertise o is discussed elsewhere.

12 The new(ish) science of people

It is likely that the formal education of this book's readership will predominantly have been based on the so-called *hard sciences*. The word *hard* being used in the sense of having theories backed by solid experimental results, which are repeatable and have been repeated many times. These sciences, and many engineering disciplines, have also been studied experimentally for a long period of time. The controversies surrounding the basic theory, taught to undergraduates, have been worked through.

Psychology has none of those advantages. There are often unseen, complex interactions going on inside the object being studied (people's responses to questions and problems). Because of this, studies using slightly different experimental situations can obtain very different results. The field is also relatively new, and the basic theory is still being argued over. Consequently, this book cannot provide a definitive account of the underlying theories relating to the subject of immediate interest here—reading and writing source code.

The results of studies, and theories, from psychology are starting to become more widely applied in other fields. For instance, economists are starting to realize that people do not always make rational decisions.^[403] Researchers are also looking at the psychology of programming.

The subfield of psychology that is of most relevance to this book is cognitive psychology. The goal of cognitive psychology is to understand the nature of human intelligence and how it works. Other subfields include clinical psychology (understanding why certain thought malfunctions occur) and social psychology (how people behave in groups or with other individuals).^{0.5}

cognitive
psychology

12.1 Brief history of cognitive psychology

Topics of interest to cognitive psychology were discussed by the Greeks as part of their philosophical thinking. This connection with philosophy continued through the works of Descartes, Kant, Mill, and others. In 1879, Wilhelm Wundt established the first psychology laboratory in Germany; this date is considered to mark the start of psychology as an independent field. Wundt believed that the workings of the mind were open to self-observation. The method involved introspection by trained observers under controlled conditions. Unfortunately, different researchers obtained different results from these introspection experiments, so the theory lost creditability.

During the 1920s, John Watson and others developed the theory known as *Behaviorism*. This theory was based on the idea that psychology should be based on external behavior, not on any internal workings of the mind. The theory is best known through its use of rats in various studies. Although widely accepted in the US for a long time, behaviorism was not so dominant in Europe, where other theories were also developed.

Measurements on human performance were given a large boost by World War II. The introduction of technology, such as radar, required people to operate it. Information about how people were best trained to use complex equipment, and how they could best maintain their attention on the job at hand, was needed.

Cognitive psychology grew into its current form through work carried out between 1950 and 1970. The inner workings of the mind were center stage again. The invention of the computer created a device, the operation of which was seen as a potential parallel for the human mind. Information theory as a way of processing information started to be used by psychologists. Another influence was linguistics, in particular Noam Chomsky's theories for analyzing the structure of language. The information-processing approach to cognitive psychology is based on carrying out experiments that measured human performance and building models that explained the results. It does not concern itself with actual processes within the brain, or parts of the brain, that might perform these functions.

Since the 1970s, researchers have been trying to create theories that explain human cognition in terms of how the brain operates. These theories are known as *cognitive architectures*. The availability of brain scanners (which enable the flow of blood through the brain to be monitored, equating blood flow to activity) in the 1990s has created the research area of cognitive neuroscience, which looks at brain structure and processes.

^{0.5}For a good introduction to the subject covering many of the issues discussed here, see either *Cognitive Psychology: A Student's Handbook* by Eysenck and Keane^[118] or *Cognitive Psychology and its Implications* by Anderson.^[11]

12.2 Evolutionary psychology

evolutionary
psychology

Human cognitive processes are part of the survival package that constitutes a human being. The cognitive processes we have today exist because they increased (or at least did not decrease) the likelihood of our ancestors passing on their genes thorough offspring. Exactly what edge these cognitive processes gave our ancestors, over those who did not possess them, is a new and growing area of research known as *evolutionary psychology*. To quote one of the founders of the field:^[83]

Cosmides^[83] *Evolutionary psychology is an approach to psychology, in which knowledge and principles from evolutionary biology are put to use in research on the structure of the human mind. It is not an area of study, like vision, reasoning, or social behavior. It is a way of thinking about psychology that can be applied to any topic within it.*

. . . all normal human minds reliably develop a standard collection of reasoning and regulatory circuits that are functionally specialized and, frequently, domain-specific. These circuits organize the way we interpret our experiences, inject certain recurrent concepts and motivations into our mental life, and provide universal frames of meaning that allow us to understand the actions and intentions of others. Beneath the level of surface variability, all humans share certain views and assumptions about the nature of the world and human action by virtue of these human universal reasoning circuits.

conjunc-
tion fallacy

These functionally specialized circuits (the theory often goes by the name of the *massive modularity hypothesis*) work together well enough to give the impression of a powerful, general purpose processor at work. Because they are specialized to perform a given task when presented with a problem that does not have the expected form (the use of probabilities rather than frequency counts in the conjunction fallacy) performance is degraded (peoples behavior appears incompetent, or even irrational, if presented with a reasoning problem). The following are the basic principles:

Cosmides^[83] *Principle 1. The brain is a physical system. It functions as a computer. Its circuits are designed to generate behavior that is appropriate to your environmental circumstances.*

Principle 2. Our neural circuits were designed by natural selection to solve problems that our ancestors faced during our species' evolutionary history.

Principle 3. Consciousness is just the tip of the iceberg; most of what goes on in your mind is hidden from you. As a result, your conscious experience can mislead you into thinking that our circuitry is simpler than it really is. Most problems that you experience as easy to solve are very difficult to solve— they require very complicated neural circuitry.

Principle 4. Different neural circuits are specialized for solving different adaptive problems.

Principle 5. Our modern skulls house a stone age mind.

Although this field is very new and has yet to establish a substantial body of experimental results and theory, it is referred to throughout these coding guidelines. The standard reference is Barkow, Cosmides, and Tooby^[34] (Mithen^[302] provides a less-technical introduction).

12.3 Experimental studies

experimental
studies

Much of the research carried out in cognitive psychology has used people between the ages of 18 and 21, studying some form of psychology degree, as their subjects. There has been discussion by psychology researchers on the extent to which these results can be extended to the general populace.^[33] However, here we are interested in the extent to which the results obtained using such subjects is applicable to how developers behave?

Given that people find learning to program difficult, and there is such a high failure rate for programming courses^[234] it is likely that some kind of ability factors are involved. However, because of the lack of studies investigating this issue, it is not yet possible to know what these programming ability factors might be. There are a large number of developers who did not study for some form of a computing degree at university, so the fact that experimental subjects are often students taking other kinds of courses is unlikely to be an issue.

12.3.1 *The importance of experiments*

The theories put forward by the established sciences are based on experimental results. Being elegant is not sufficient for a theory to be accepted; it has to be backed by experiments.

Software engineering abounds with theories, and elegance is often cited as an important attribute. However, experimental results for these theories are often very thin on the ground. The computing field is evolving so rapidly that researchers do not seem willing to invest significant amounts of their time gathering experimental data when there is a high probability that many of the base factors will have completely changed by the time the results are published.

Replication is another important aspect of scientific research; others should be able to duplicate the results obtained in the original experiment. Replication of experiments within software research is relatively rare; possible reasons include

- the pace of developments in computing means that there are often more incentives for trying new ideas rather than repeating experiments to verify the ideas of others,
- the cost of performing an experiment can be sufficiently high that the benefit of replication is seen as marginal, and/or
- the nature of experiments involving large-scale, commercial projects are very difficult to replicate. Source code can be duplicated perfectly, so there is no need to rewrite the same software again.

A good practical example of the benefits of replication and the dangers of not doing any is given by Brooks.^[54] Another important issue is the statistical power of experiments.^[297] Experiments that fail can be as important as those that succeed. Nearly all published, computing-related papers describe successes. The benefits of publishing negative results (i.e., ideas that did not work) has been proposed by Prechelt.^[361] A study^[416] of 5,453 papers published in software engineering journals between 1993 and 2002 found that only 1.9% reported controlled experiments (of which 72.6% used students only as subjects) and even then the statistical power of these experiments fell below expected norms.^[103]

12.4 **The psychology of programming**

Studies on the psychology of programming have taken their lead from trends in both psychology and software engineering. In the 1960s and 1970s, studies attempted to measure performance times for various tasks. Since then researchers have tried to build models of how people carry out the tasks involved with various aspects of programming.

psychology of programming

Several theories about how developers go about the task of comprehending source code have been proposed. There have also been specific proposals about how to reduce developer error rates, or to improve developer performance. Unfortunately, the experimental evidence for these theories and proposals is either based on the use of inexperienced subjects or does not include sufficient data to enable statistically significant conclusions to be drawn. A more detailed, critical analysis of the psychological study of programming is given by Sheil^[399] (the situation does not seem to have changed since this paper was written 20 years ago).

Several studies have investigated how novices write software. This is both an area of research interest and of practical use in a teaching environment. The subjects taking part in these studies also have the characteristics of the population under investigation (i.e., predominantly students). However, this book is aimed at developers who have several years experience writing code; it is not aimed at novices and it does not teach programming skills.

Lethbridge, Sim, and Singer^[256] discuss some of the techniques used to perform field studies inside software companies.

12.4.1 *Student subjects*

Although cognitive psychology studies use university students as their subjects there is an important characteristic they generally have, for these studies, that they don't have for software development studies.^[415] That characteristic is experience— that is, years of practice performing the kinds of actions (e.g., reading text,

making decisions, creating categories, reacting to inputs) they are asked to carry out in the studies. However, students, typically, have very little experience of writing software, perhaps 50 to 150 hours. Commercial software developers are likely to have between 1,000 to 10,000 hours of experience. A study by Moher and Schneider^[306] compared the performance of students and professional developers in program comprehension tasks. The results showed that experience was a significant predictor of performance level (greater than aptitude in this study).

Reading and writing software is a learned skill. Any experiments that involve a skill-based performance need to take into account the subjects' skill level. The coding guidelines in this book are aimed at developers in a commercial environment where it is expected that they will have at least two years experience in software development.

Use of very inexperienced developers as subjects in studies means that there is often a strong learning effect in the results. Student subjects taking part in an experiment often get better at the task because they are learning as they perform it. Experienced developers have already acquired the skill in the task being measured, so there is unlikely to be any significant learning during the experiment. An interesting insight into the differences between experiments involving students and professional developers is provided by a study performed by Basili^[35] and a replication of it by Ciolkowski.^[74]

A note on differences in terminology needs to be made here. Many studies in the psychology of programming use the phrase *expert* to apply to a subject who is a third-year undergraduate or a graduate student (the term *novice* being applied to first-year undergraduates). In a commercial software development environment a recent graduate is considered to be a *novice* developer. Somebody with five or more years of commercial development experience might know enough to be called an *expert*.

12.4.2 Other experimental issues

When an experiment is performed, it is necessary to control all variables except the one being measured. It is also necessary to be able to perform the experiments in a reasonable amount of time. Most commercial programs contain thousands of lines of source code. Nontrivial programs of this size can contain any number of constructs that could affect the results of an experiment; they would also require a significant amount of effort to read and comprehend. Many experiments use programs containing less than 100 lines of source. In many cases, it is difficult to see how results obtained using small programs will apply to much larger programs.

The power of the statistical methods used to analyze experimental data depends on the number of different measurements made. If there are few measurements, the statistical significance of any claim's results will be small. Because of time constraints many experiments use a small number of different programs, sometimes a single program. All that can be said for any results obtained for a single program is that the results apply to that program; there is no evidence of generalization to any other programs.

Is the computer language used in experiments significant? The extent to which the natural language, spoken by a person, affects their thinking has been debated since Boas, Sapir, and Whorf developed the linguistic relativity hypothesis^[268]. In this book, we are interested in C, a member of the procedural computer language family. More than 99.9% of the software ever written belongs to languages in this family. However, almost as many experiments seem to use nonprocedural languages, as procedural ones. Whether the language family of the experiment affects the applicability of the results to other language families is unknown. However, it will have an effect on the degree of believability given to these results by developers working in a commercial environment.

12.5 What question is being answered?

Many of the studies carried out by psychologists implicitly include a human language (often English) as part of the experiment. Unless the experiments are carefully constructed, unexpected side-effects may be encountered. These can occur because of the ambiguous nature of words in human language, or because of subjects expectations based on their experience of the nature of human communication.

The following three subsections describe famous studies, which are often quoted in introductory cognitive psychology textbooks. Over time, these experiments have been repeated in various, different ways and the

underlying assumptions made by the original researchers has been challenged. The lesson to be learned from these studies is that it can be very difficult to interpret a subject's answer to what appears to be a simple question. Subjects simply may not have the intellectual machinery designed to answer the question in the fashion it is phrased (base rate neglect), they may be answering a completely different question (conjunction fallacy), or they may be using a completely unexpected method to solve a problem (availability heuristic).

12.5.1 Base rate neglect

Given specific evidence, possible solutions to a problem can be ordered by the degree to which they are representative of that evidence (i.e., their probability of occurring as the actual solution, based on past experience). While these representative solutions may appear to be more likely to be correct than less-representative solutions, for particular cases they may in fact be less likely to be the solution. Other factors, such as the prior probability of the solution, and the reliability of the evidence can affect the probability of any solution being correct.

base rate neglect
representative heuristic

A series of studies, Kahneman and Tversky^[211] suggested that subjects often seriously undervalue the importance of prior probabilities (i.e., they neglected base-rates). The following is an example from one of these studies. Subjects were divided into two groups, with one group of subjects being presented with the following cover story:

A panel of psychologists have interviewed and administered personality tests to 30 engineers and 70 lawyers, all successful in their respective fields. On the basis of this information, thumbnail descriptions of the 30 engineers and 70 lawyers have been written. You will find on your forms five descriptions, chosen at random from the 100 available descriptions. For each description, please indicate your probability that the person described is an engineer, on a scale from 0 to 100.

and the other group of subjects presented with identical cover story, except the prior probabilities were reversed (i.e., they were told that the personality tests had been administered to 70 engineers and 30 lawyers). Some of the descriptions provided were designed to be compatible with the subjects' stereotype of engineers, others were designed to be compatible with the stereotypes of lawyers, and one description was intended to be neutral. The following are two of the descriptions used.

Jack is a 45-year-old man. He is married and has four children. He is generally conservative, careful and ambitious. He shows no interest in political and social issues and spends most of his free time on his many hobbies which include home carpentry, sailing, and mathematical puzzles.
The probability that Jack is one of the 30 engineers in the sample of 100 is ____%.

Dick is a 30-year-old man. He is married with no children. A man of high ability and high motivation, he promises to be quite successful in his field. He is well liked by his colleagues.
The probability that Dick is one of the 70 lawyers in the sample of 100 is ____%.

Following the five descriptions was this null description.

Suppose now that you are given no information whatsoever about an individual chosen at random from the sample.
The probability that this man is one of the 30 engineers in the sample of 100 is ____%.

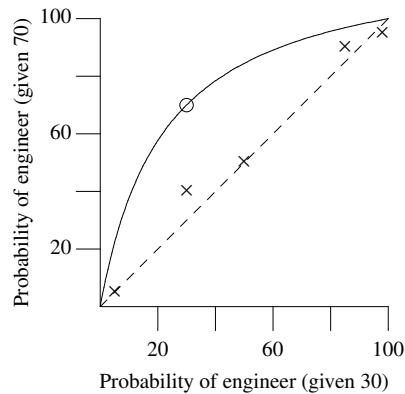


Figure 0.10: Median judged probability of subjects choosing an engineer, for five descriptions and for the null description (unfilled circle symbol). Adapted from Kahneman.^[211]

In both groups, half of the subjects were asked to evaluate, for each description, if the person described was an engineer. The other subjects were asked the same question, except they were asked about lawyers.

The probability of a person being classified as an engineer, or lawyer, can be calculated using Bayes' theorem. Assume that, after reading the description, the estimated probability of that person being an engineer is P . The information that there are 30 engineers and 70 lawyers in the sample allows us to modify the estimate, P , to obtain a more accurate estimate (using all the information available to us). The updated probability is $0.3P/(0.3P + 0.7(1 - P))$. If we are told that there are 70 engineers and 30 lawyers, the updated probability is $0.7P/(0.7P + 0.3(1 - P))$. For different values of the estimate P , we can plot a graph using the two updated probabilities as the x and y coordinates. If information on the number of engineers and lawyers is not available, or ignored, the graph is a straight line.

The results (see Figure 0.10) were closer to the straight line than the Bayesian line. The conclusion drawn was that information on the actual number of engineers and lawyers in the sample (the base-rate) had minimal impact on the subjective probability chosen by subjects.

Later studies^[229] found that peoples behavior when making decisions that included a base-rate component was complex. Use of base-rate information was found to depend on how problems and the given information was framed (large between study differences in subject performance were also seen). For instance, in some cases subjects were found to use their own experiences to judge the likelihood of certain events occurring rather than the probabilities given to them in the studies. In some cases the ecological validity of using Bayes' theorem to calculate the probabilities of outcomes has been questioned.

To summarize: while people have been found to ignore base-rates when making some decisions, this behavior is far from being universally applied to all decisions.

12.5.2 The conjunction fallacy

An experiment originally performed by Tversky and Kahneman^[464] presented subjects with the following problem.

Linda is 31 years old, single, outspoken, and very bright. She majored in philosophy. As a student, she was deeply concerned with issues of discrimination and social justice, and also participated in anti-nuclear demonstrations.

Please rank the following statements by their probability, using 1 for the most probable and 8 for the least probable.

- Linda is a teacher in elementary school.
- Linda works in a bookstore and takes Yoga classes.

- (c) Linda is active in the feminist movement.
- (d) Linda is a psychiatric social worker.
- (e) Linda is a member of the League of Women Voters.
- (f) Linda is a bank teller.
- (g) Linda is an insurance sales person.
- (h) Linda is a bank teller and is active in the feminist movement.

In a group of subjects with no background in probability or statistics, 89% judged that statement (h) was more probable than statement (f). Use of simple mathematical logic shows that Linda cannot be a feminist bank teller unless she is also a bank teller, implying that being only a bank teller is at least as likely, if not more so, than being both a bank teller and having some additional attribute. When the subjects were graduate students in the decision science program of the Stanford Business School (labeled as statistically sophisticated by the experimenters), 85% judged that statement (h) was more probable than statement (f).

These results (a compound event being judged more probable than one of its components) have been duplicated by other researchers performing different experiments. A recent series of studies^[406] went as far as checking subjects' understanding of the word *probability* and whether statement (f) might be interpreted to mean *Linda is a bank teller and not active in the feminist movement* (it was not).

This pattern of reasoning has become known as *the conjunction fallacy*.

On the surface many of the subjects in the experiment appear to be reasoning in a nonrational way. How can the probability of the event *A and B* be greater than the probability of event *A*? However, further studies have found that the likelihood of obtaining different answers can be affected by how the problem is expressed. The effects of phrasing the problem in terms of either *probability* or *frequency* were highlighted in a study by Fiedler.^[128] The original Tversky and Kahneman study wording was changed to the following:

- There are 100 people who fit the description above. How many of them are:
- (a) bank tellers?
 - (b) bank tellers and active in the feminist movement?
 - ...

In this case, only 22% of subjects rated the *bank teller and active in the feminist movement* option as being more frequent than the *bank teller* only option. When Fiedler repeated the experiment using wording identical to the original Tversky and Kahneman experiment, 91% of subjects gave the feminist bank teller option as more probable than the bank teller only option. A number of different explanations, for the dependence of the conjunction fallacy on the wording of the problem, have been proposed.

Evolutionary psychologists have interpreted these results as showing that people are not very good at reasoning using probability. It is argued that, in our daily lives, events are measured in terms of their frequency of occurrence (e.g., how many times fish were available at a particular location in the river). This event-based measurement includes quantity, information not available when probabilities are used. Following this argument through suggests that the human brain has become specialized to work with frequency information, not probability information.

Hertwig and Gigerenzer^[168] point out that, in the Linda problem, subjects were not informed that they were taking part in an exercise in probability. Subjects therefore had to interpret the instructions; in particular, what did the experimenter mean by *probability*? Based on Grice's^[151] theory of conversational reasoning, they suggested that the likely interpretation given to the word *probability* would be along the lines of "something which, judged by present evidence, is likely to be true, to exist, or to happen," (one of the Oxford English dictionary contemporary definitions of the word), not the mathematical definition of the word.

Grice's theory was used to make the following predictions:

Prediction 1: Probability judgments. If asked for probability judgments, people will infer its nonmathematical meanings, and the proportion of conjunction violations will be high as a result.

Prediction 2: Frequency judgments. If asked for frequency judgments, people will infer mathematical meanings, and the proportion of conjunction violations will decrease as a result.

Prediction 3: Believability judgments. If the term “probability” is replaced by “believability”, then the proportion of conjunction violations should be about as prevalent as in the probability judgment.

A series of experiments confirmed these predictions. A small change in wording caused subjects to have a completely different interpretation of the question.

12.5.3 Availability heuristic

availability heuristic

How do people estimate the likelihood of an occurrence of an event? The availability heuristic argues that, in making an estimate, people bring to mind instances of the event; the more instances brought to mind, the more likely it is to occur. Tversky and Kahneman^[462] performed several studies in an attempt to verify that people use this heuristic to estimate probabilities. Two of the more well-known experiments follow.

The first is judgment of word frequency; here subjects are first told that.

The frequency of appearance of letters in the English language was studied. A typical text was selected, and the relative frequency with which various letters of the alphabet appeared in the first and third positions in words was recorded. Words of less than three letters were excluded from the count.

You will be given several letters of the alphabet, and you will be asked to judge whether these letters appear more often in the first or in the third position, and to estimate the ratio of the frequency with which they appear in these positions.

They were then asked the same question five times, using each of the letters (K, L, N, R, V).

Consider the letter R.

Is R more likely to appear in:

- the first position?
- the third position? (check one)

My estimate for the ratio of these two values is ____:1.

Of the 152 subjects, 105 judged the first position to be more likely (47 the third position more likely). The median estimated ratio was 2:1.

In practice, words containing the letter *R* in the third position occur more frequently in texts than words with *R* in the first position. This is true for all the letters—*K, L, N, R, V*.

The explanation given for these results was that subjects could more easily recall words beginning with the letter *R*, for instance, than recall words having an *R* as the third letter. The answers given, being driven by the availability of instances that popped into the subjects’ heads, not by subjects systematically counting all the words they knew.

An alternative explanation of how subjects might have reached their conclusion was proposed by Sedlmeier, Hertwig, and Gigerenzer.^[396] First they investigated possible ways in which the availability heuristic might operate; Was it based on availability-by-number (the number of instances that could be recalled) or availability-by-speed (the speed with which instances can be recalled). Subjects were told (the following is an English translation, the experiment took place in Germany and used German students) either:

Your task is to recall as many words as you can in a certain time. At the top of the following page you will see a letter. Write down as many words as possible that have this letter as the first (second) letter.

or,

Your task is to recall as quickly as possible one word that has a particular letter as the first (second) letter. You will hear first the position of the letter and then the letter. From the moment you hear the letter, try to recall a respective word and verbalize this word.

Subjects answers were used to calculate an estimate of relative word frequency based on either availability-by-number or on availability-by-speed. These relative frequencies did not correlate with actual frequency of occurrence of words in German. The conclusion drawn was that the availability heuristic was not an accurate estimator of word frequency, and that it could not be used to explain the results obtained by Tversky and Kahneman.

If subjects were not using either of these availability heuristics, what mechanism are they using? Jonides and Jones^[205] have shown, based on a large body of results, that subjects are able to judge the number of many kinds of events in a way that reflects the actual relative frequencies of the events with some accuracy.

Sedlmeier et al.^[396] proposed (what they called the *regressed-frequencies hypothesis*) that (a) the frequencies with which individual letters occur at different positions in words are monitored (by people while reading), and (b) the letter frequencies represented in the mind are regressed toward the mean of all letter frequencies. This is a phenomenon often encountered in frequency judgment tasks, where low frequencies tend to be overestimated and high frequencies underestimated; although this bias affects the accuracy of the absolute size of frequency judgments, it does not affect their rank order. Thus, when asked for the relative frequency of a particular letter, subjects should be expected to give judgments of relative letter frequencies that reflect the actual ones, although they will overestimate relative frequencies below the mean and underestimate those above the mean — a simple regressed-frequency heuristic. The studies performed by Sedlmeier et al. consistently showed subjects' judgments conforming best to the predictions of the regressed-frequencies hypothesis.

While it is too soon to tell if the regressed-frequencies hypothesis is the actual mechanism used by subjects, it does offer a better fit to experimental results than the availability heuristic.

13 Categorization

Children as young as four have been found to use categorization to direct the inferences they make,^[140] and many different studies have shown that people have an innate desire to create and use categories (they have also been found to be sensitive to the costs and benefits of using categories^[274]). By dividing items in the world into categories of things, people reduce the amount of information they need to learn^[360] by building an indexed data structure that will enable them to lookup information on specific items they may not have encountered before (by assigning that item to one or more categories and extracting information common to items in those categories). For instance, a flying object with feathers and a beak might be assigned to the category *bird*, which suggests the information that it lays eggs and may be migratory.

Source code is replete with examples of categories; similar functions are grouped together in the same source file, objects belonging to a particular category are defined as members of the same structure type, and enumerated types are defined to represent a common set of symbolic names.

People seem to have an innate desire to create categories (people have been found to expect random sequences to have certain attributes,^[120] e.g., frequent alternation between different values, which from a mathematical perspective represent regularity). There is the danger that developers, reading a program's source code will create categories that the original author was not aware existed. These *new* categories may

categorization

translation unit
syntax
structure type
sequentially
allocated objects
declaration
syntax
enumeration
set of named
constants
symbolic
name

represent insights into the workings of a program, or they may be completely spurious (and a source of subsequent incorrect assumptions, leading to faults being introduced).

Categories can be used in many thought processes without requiring significant cognitive effort (a built-in operation). For instance, categorization can be used to perform inductive reasoning (the derivation of generalized knowledge from specific instances), and to act as a memory aid (remembering the members of a category). There is a limit on the cognitive effort that developers have available to be used and making use of a powerful ability, which does not require a lot of effort, helps optimize the use of available resources.

There have been a number of studies^[379] looking at how people use so-called *natural categories* (i.e., those occurring in nature such as mammals, horses, cats, and birds) to make inductive judgments. People's use of categorical-based arguments (e.g., "Grizzly bears love onions." and "Polar bears love onions." therefore "All bears love onions.") has also been studied.^[332]

Source code differs from nature in that it is created by people who have control over how it is organized. Recognizing that people have an innate ability to create and use categories, there is a benefit in trying to maximize positive use (developers being able to infer probable behaviors and source code usage based on knowing small amounts of information) of this ability and to minimize negative use (creating unintended categories, or making inapplicable inductive judgments).

Source code can be organized in a myriad of ways. The problem is finding the optimal organization, which first requires knowing what needs to be optimized. For instance, I might decide to split some functions I have written that manipulate matrices and strings into two separate source files. I could decide that the functions I wrote first will go in the first file and those that I wrote later in the second file, or perhaps the first file will contain those functions used on project X and the second file those functions used on project Y. To an outside observer, a more *natural* organization might be to place the matrix-manipulation functions in the first file and the string-manipulation functions in the second file.

In a project that grows over time, functions may be placed in source files on an as-written basis; a maintenance process that seeks to minimize disruption to existing code will keep this organization. When two separate projects are merged into one, a maintenance process that seeks to minimize disruption to existing code is unlikely to reorganize source file contents based on the data type being manipulated. This categorization process, based on past events, is a major factor in the difficulty developers have in comprehending *old* source. Because category membership is based on historical events, developers either need knowledge of those events or they have to memorize information on large quantities of source. Program comprehension changes from using category-based induction to relying on memory for events or source code.

Even when the developer is not constrained by existing practices the choice of source organization is not always clear-cut. An organization based on the data type being manipulated is one possibility, or there may only be a few functions and an organization based on functionality supported (i.e., printing) may be more appropriate. Selecting which to use can be a difficult decision. The following subsections discuss some of the category formation studies that have been carried out, some of the theories of category formation, and possible methods of calculating similarity to category.

Situations where source code categorization arise include: deciding which structure types should contain which members, which source files should contain which object and function definitions, which source files should be kept in which directories, whether functionality should go in a single function or be spread across several functions, and what is the sequence of identifiers in an enumerated type?

Explicitly organizing source code constructs so that future readers can make use of their innate ability to use categories, to perform inductive reasoning, is not meant to imply that other forms of reasoning are not important. The results of deductive reasoning are generally the norm against which developer performance is measured. However, in practice, developers do create categories and use induction. Coding guidelines need to take account of this human characteristic. Rather than treating it as an aberration that developers need to be trained out of, these coding guidelines seek to adapt to this innate ability.

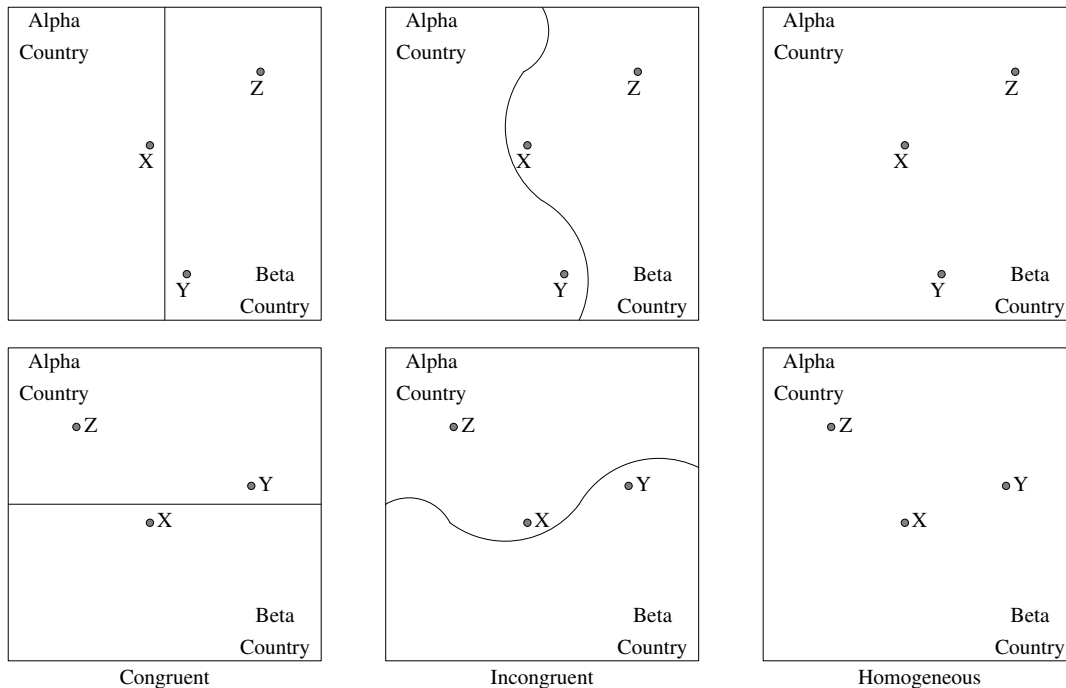


Figure 0.11: Country boundaries distort judgment of relative city locations. Adapted from Stevens.^[431]

13.1 Category formation

How categories should be defined and structured has been an ongoing debate within all sciences. For instance, the methods used to classify living organisms into family, genus, species, and subspecies has changed over the years (e.g., most recently acquiring a genetic basis).

Categories do not usually exist in isolation. Category judgment is often organized according to a hierarchy of relationships between concepts—a taxonomy. For instance, Jack Russell, German Shepherd, and Terrier belong to the category of dog, which in turn belongs to the category of mammal, which in turn belongs to the category of living creature. Organizing categories into hierarchies means that an attribute of a higher-level category can affect the perceived attributes of a subordinate category. This effect was illustrated in a study by Stevens and Coupe.^[431] Subjects were asked to remember the information contained in a series of maps (see Figure 0.11). They were then asked questions such as: “Is X east or west of Y?”, and “Is X north or south of Y?” Subjects gave incorrect answers 18% of the time for the congruent maps, but 45% of the time for the incongruent maps (15% for the homogeneous). They were using information about the relative locations of the countries to answer questions about the city locations.

Several studies have shown that people use around three levels of abstraction in creating hierarchical relationships. Rosch^[385] called the highest level of abstraction the *superordinate-level*—for instance, the general category furniture. The next level down is the *basic-level*; this is the level at which most categorization is carried out—for instance, car, truck, chair, or table. The lowest level is the *subordinate-level*, denoting specific types of objects. For instance, a family car, a removal truck, my favourite armchair, a kitchen table. Rosch found that the basic-level categories had properties not shared by the other two categories; adults spontaneously name objects at this level. It is also the abstract level that children acquire first, and category members tend to have similar overall shapes.

- A study by Markman and Wisniewski^[277] investigated how people view superordinate-level and basic-level categories as being different. The results showed that basic-level categories, derived from the

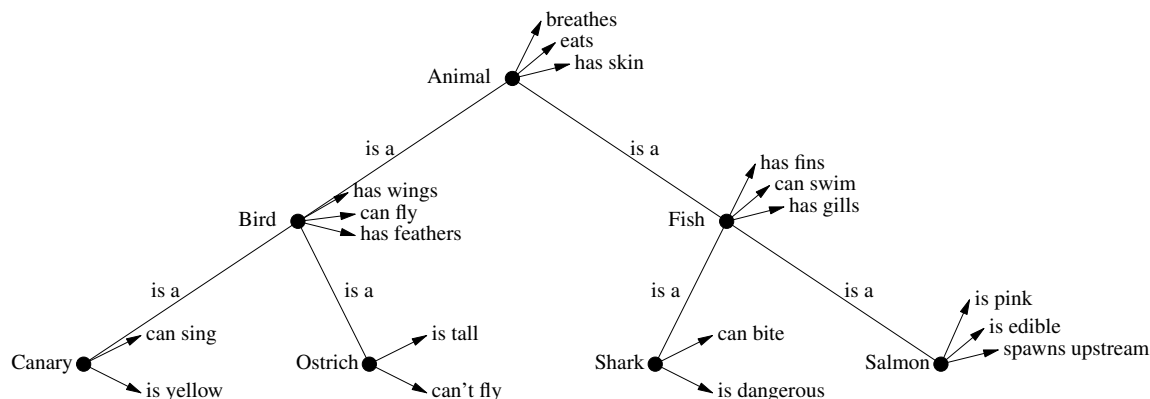


Figure 0.12: Hypothetical memory structure for a three-level hierarchy. Adapted from Collins.^[78]

same superordinate-level, had a common structure that made it easy for people to compare attributes; for instance, motorcycle, car, and truck are basic-level categories of vehicle. They all share attributes (so-called *alignable differences*), for instance, number of wheels, method of steering, quantity of objects that can be carried, size of engine, and driver qualifications that differ but are easily compared. Superordinate-level categories differ from each other in that they do not share a common structure. This lack of a common structure means it is not possible to align their attributes to differentiate them. For these categories, differentiation occurs through the lack of a common structure. For instance, the superordinate-level categories — vehicle, musical instrument, vegetable, and clothing — do not share a common structure.

- A study by Tanaka and Taylor^[442] showed that the quantity of a person’s knowledge and experience can affect the categories they create and use.
- A study by Johansen and Palmeri^[200] showed that representations of perceptual categories can change with categorization experience. While these coding guidelines are aimed at experienced developers, they recognize that many experienced developers are likely to be inexperienced comprehenders of much of the source code they encounter. The guidelines in this book take the default position that, given a choice, they should assume an experienced developer who is inexperienced with the source being read.

There are likely to be different ways of categorizing the various components of source code. These cases are discussed in more detail elsewhere. Commonality and regularities shared between different sections of source code may lead developers to implicitly form categories that were not intended by the original authors. The extent to which the root cause is poor categorization by the original developers, or simply unrelated regularities, is not discussed in this book.

What method do people use to decide which, if any, category a particular item is a member of? Several different theories have been proposed and these are discussed in the following subsections.

13.1.1 The Defining-attribute theory

The defining-attribute theory proposes that members of a category are characterized by a set of defining attributes. This theory predicts that attributes should divide objects up into different concepts whose boundaries are well defined. All members of the concept are equally representative. Also, concepts that are a basic-level of a superordinate-level concept will have all the attributes of that superordinate level; for instance, a sparrow (small, brown) and its superordinate bird (two legs, feathered, lays eggs); see Figure 0.12.

Although scientists and engineers may create and use defining-attribute concept hierarchies, experimental evidence shows that people do not naturally do so. Studies have shown that people do not treat category

members as being equally representative, and some are rated as more typical than others.^[380] Evidence that people do not structure concepts into the neat hierarchies required by the defining-attribute theory was provided by studies in which subjects verified membership of a more distant superordinate more quickly than an immediate superordinate (according to the theory, the reverse situation should always be true).

13.1.2 The Prototype theory

In this theory, categories have a central description, the prototype, that represents the set of attributes of the category. This set of attributes need not be necessary, or sufficient, to determine category membership. The members of a category can be arranged in a typicality gradient, representing the degree to which they represent a typical member of that category. It is also possible for objects to be members of more than one category (e.g., tomatoes as a fruit, or a vegetable).

13.1.3 The Exemplar-based theory

The exemplar-based theory of classification proposes that specific instances, or *exemplars*, act as the prototypes against which other members are compared. Objects are grouped, relative to one another, based on some similarity metric. The exemplar-based theory differs from the prototype theory in that specific instances are the norm against which membership is decided. When asked to name particular members of a category, the attributes of the exemplars are used as cues to retrieve other objects having similar attributes.

13.1.4 The Explanation-based theory

The explanation-based theory of classification proposes that there is an explanation for why categories have the members they do. For instance, the biblical classification of food into *clean* and *unclean* is roughly explained by saying that there should be a correlation between type of habitat, biological structure, and form of locomotion; creatures of the sea should have fins, scales, and swim (sharks and eels don't) and creatures of the land should have four legs (ostriches don't).

From a predictive point of view, explanation-based categories suffer from the problem that they may heavily depend on the knowledge and beliefs of the person who formed the category; for instance, the set of objects a person would remove from their home while it was on fire.

Murphy and Medin^[313] discuss how people can use explanations to achieve conceptual coherence in selecting the members of a category (see Table 0.5).

Table 0.5: General properties of explanations and their potential role in understanding conceptual coherence. Adapted from Murphy.^[313]

Properties of Explanations	Role in Conceptual Coherence
<i>Explanation</i> of a sort, specified over some domain of observation	Constrains which attributes will be included in a concept representation Focuses on certain relationships over others in detecting attribute correlations
Simplify reality	Concepts may be idealizations that impose more structure than is <i>objectively</i> present
Have an external structure— fits in with (or do not contradict) what is already known	Stresses intercategory structure; attributes are considered essential to the degree that they play a part in related theories (external structures)
Have an internal structure— defined in part by relations connecting attributes	Emphasizes mutual constraints among attributes. May suggest how concept attributes are learned
Interact with data and observations in some way	Calls attention to inference processes in categorization and suggests that more than attribute matching is involved

13.2 Measuring similarity

The intent is for these guideline recommendations to be automatically enforceable. This requires an algorithm for calculating similarity, which is the motivation behind the following discussion.

How might two objects be compared for similarity? For simplicity, the following discussion assumes an object can have one of two values for any attribute, yes/no. The discussion is based on material in

Classification and Cognition by W. K. Estes.^[116]

similarity
product rule

To calculate the similarity of two objects, their corresponding attributes are matched. The product of the similarity coefficient of each of these attributes is computed. A matching similarity coefficient, t (a value in the range one to infinity, and the same for every match), is assigned for matching attributes. A nonmatching similarity coefficient, s_i (a value in the range 0 to 1, and potentially different for each nonmatch), is assigned for each nonmatching coefficient. For example, consider two birds that either have (plus sign), or do not have (minus sign), some attribute (numbered 1 to 6 in Table 0.6). Their similarity, based on these attributes is $t \times t \times s_3 \times t \times s_5 \times t$.

Table 0.6: Computation of pattern similarity. Adapted from Estes.^[116]

Attribute	1	2	3	4	5	6
Starling	+	+	-	+	+	+
Sandpiper	+	+	+	+	-	+
Attribute similarity	t	t	s_3	t	s_5	t

When comparing objects within the same category the convention is to give the similarity coefficient, t , for matching attributes, a value of one. Another convention is to give the attributes that differ the same similarity coefficient, s . In the preceding case, the similarity becomes s^2 .

Sometimes the similarity coefficient for matches needs to be taken into account. For instance, in the following two examples the similarity between the first two character sequences is ts , while in the second is t^3s . Setting t to be one would result in both pairs of character sequences being considered to have the same similarity, when in fact the second sequence would be judged more similar than the first. Studies on same/different judgments show that both reaction time and error rates increase as a function of the number of items being compared.^[238] The value of t cannot always be taken to be unity.

A	B	A	B	C	D
A	E	A	E	C	D

The previous example computed the similarity of two objects to each other. If we have a category, we can calculate a similarity to category measure. All the members of a category are listed. The similarity of each member, compared with every other member, is calculated in turn and these values are summed for that member. Such a calculation is shown in Table 0.7.

Table 0.7: Computation of similarity to category. Adapted from Estes.^[116]

Object	Ro	Bl	Sw	St	Vu	Sa	Ch	Fl	Pe	Similarity to Category
Robin	1	1	1	s	s^4	s	s^5	s^6	s^5	$3 + 2s + s^4 + 2s^5 + s^6$
Bluebird	1	1	1	s	s^4	s	s^5	s^6	s^5	$3 + 2s + s^4 + 2s^5 + s^6$
Swallow	1	1	1	s	s^4	s	s^5	s^6	s^5	$3 + 2s + s^4 + 2s^5 + s^6$
Starling	s	s	s	1	s^3	s^2	s^6	s^5	s^6	$1 + 3s + s^2 + s^3 + s^5 + 2s^6$
Vulture	s^4	s^4	s^4	s^3	1	s^5	s^3	s^2	s^3	$1 + s^2 + 3s^3 + 3s^4 + s^5$
Sandpiper	s	s	s	s^2	s^5	1	s^4	s^5	s^4	$1 + 3s + s^2 + s^4 + s^5$
Chicken	s^5	s^5	s^5	s^6	s^3	s^4	1	s	1	$2 + s + s^3 + s^4 + 3s^5 + s^6$
Flamingo	s^6	s^6	s^6	s^5	s^2	s^5	s	1	s	$1 + 2s + s^2 + 2s^5 + 3s^6$
Penguin	s^5	s^5	s^5	s^6	s^3	s^4	1	s	1	$2 + s + s^3 + s^4 + 3s^5 + s^6$

Some members of a category are often considered to be more typical of that category than other members. These typical members are sometimes treated as exemplars of that category, and serve as reference points when people are thinking about that category. While there is no absolute measure of typicality, it is possible to compare the typicality of two members of a category. The relative typicality, within a category for two or more objects is calculated from their ratios of similarity to category. For instance, taking the value of s as

0.5, the relative typicality of Robin with respect to Vulture is $4.14/(4.14 + 1.84) = 0.69$, and the relative typicality of Vulture with respect to Robin is $1.84/(4.14 + 1.84) = 0.31$.

It is also possible to create derived categories from existing categories; for instance, large and small birds. For details on how to calculate typicality within those derived categories, see Estes^[116] (which also provides experimental results).

An alternative measure of similarity is the contrast model. This measure of similarity depends positively on the number of attributes two objects have in common, but negatively on the number of attributes that belong to one but not the other.

similarity
contrast model

$$\text{Contrast Sim}_{12} = af(F_{12}) - bf(F_1) - cf(F_2) \quad (0.14)$$

where F_{12} is the set of attributes common to objects 1 and 2, F_1 the set of attributes that object 1 has but not object 2, and F_2 the set of attributes that object 2 has but not object 1. The quantities a , b , and c are constants. The function f is some metric based on attributes; the one most commonly appearing in published research is a simple count of attributes.

Taking the example given in Table 0.7, there are four features shared by the starling and sandpiper and one that is unique to each of them. This gives:

$$\text{Contrast Sim} = 4a - 1b - 1c \quad (0.15)$$

based on bird data we might take, for instance, $a = 1$, $b = 0.5$, and $c = 0.25$ giving a similarity of 3.25.

On the surface, these two models appear to be very different. However, some mathematical manipulation shows that the two methods of calculating similarity are related.

$$\text{Sim}_{12} = t^{n_{12}} s^{n_1 + n_2} = t^{n_{12}} s^{n_1} s^{n_2} \quad (0.16)$$

Taking the logarithm:

$$\log(\text{Sim}_{12}) = n_{12} \log(t) + n_1 \log(s) + n_2 \log(s) \quad (0.17)$$

letting $a = \log(t)$, $b = \log(s)$, $c = \log(s)$, and noting that the value of s is less than 1, we get:

$$\log(\text{Sim}_{12}) = a(n_{12}) - b(n_1) - c(n_2) \quad (0.18)$$

This expression for product similarity has the same form as the expression for contrast similarity. Although b and c have the same value in this example, in a more general form the values of s could be different.

13.2.1 Predicting categorization performance

Studies^[385] have shown that the order in which people list exemplars of categories correlates with their relative typicality ratings. These results lead to the idea that relative typicality ratings could be interpreted as probabilities of categorization responses. However, the algorithm for calculating similarity to category values does not take into account the number of times a subject has encountered a member of the category (which will control the strength of that member's entry in the subject's memory).

categorization
performance
predicting

For instance, based on the previous example of bird categories when asked to "name the bird which comes most quickly to mind, Robin or Penguin", the probability of Robin being the answer is $4.14/(4.14 + 2.80) = 0.60$, an unrealistically low probability. If the similarity values are weighted according to the frequency

of each member's entry in a subject's memory array (Estes estimated the figures given in Table 0.8), the probability of Robin becomes $1.24 / (1.24 + 0.06) = 0.954$, a much more believable probability. The need to use frequency weightings to calculate a weighted similarity value has been verified by Nosofsky.^[327]

Table 0.8: Computation of weighted similarity to category. From Estes.^[116]

Object	Similarity Formula	$s = 0.5$	Relative Frequency	Weighted Similarity
Robin	$3 + 2s + s^4 + 2s^5 + s^6$	4.14	0.30	1.24
Bluebird	$3 + 2s + s^4 + 2s^5 + s^6$	4.14	0.20	0.83
Swallow	$3 + 2s + s^4 + 2s^5 + s^6$	4.14	0.10	0.41
Starling	$1 + 3s + s^2 + s^3 + s^5 + 2s^6$	2.94	0.15	0.44
Vulture	$1 + s^2 + 3s^3 + 3s^4 + s^5$	1.84	0.02	0.04
Sandpiper	$1 + 3s + s^2 + s^4 + s^5$	2.94	0.05	0.15
Chicken	$2 + s + s^3 + s^4 + 3s^5 + s^6$	2.80	0.15	0.42
Flamingo	$1 + 2s + s^2 + 2s^5 + 3s^6$	2.36	0.01	0.02
Penguin	$2 + s + s^3 + s^4 + 3s^5 + s^6$	2.80	0.02	0.06

The method of measuring similarity just described has been found to be a good predictor of the error probability of people judging which category a stimulus belongs to. The following analysis is based on a study performed by Shepard, Hovland, and Jenkins.^[400]

A simpler example than the bird category is used to illustrate how the calculations are performed. Here, the object attributes are color and shape, made up of the four combinations black/white, triangles/squares. Taking the case where the black triangle and black square have been assigned to category A, and the white triangle and white square have been assigned to category B, we get Table 0.9.

Table 0.9: Similarity to category (black triangle and black square assigned to category A; white triangle and white square assigned to category B).

Stimulus	Similarity to A	Similarity to B
Dark triangle	$1 + s$	$s + s^2$
Dark square	$1 + s$	$s + s^2$
Light triangle	$s + s^2$	$1 + s$
Light square	$s + s^2$	$1 + s$

If a subject is shown a stimulus that belongs in category A, the expected probability of them assigning it to that category is:

$$\frac{1 + s}{(1 + s) + (s + s^2)} \Rightarrow \frac{1}{1 + s} \tag{0.19}$$

When s is 1 the expected probability is no better than a random choice; when s is 0 the probability is a certainty.

Assigning different stimulus to different categories can change the expected response probability; for instance, by assigning the black triangle and the white square to category A and assigning the white triangle and black square to category B, we get the category similarities shown in Table 0.10.

Table 0.10: Similarity to category (black triangle and white square assigned to category A; white triangle and black square assigned to category B).

Stimulus	Similarity to A	Similarity to B
Dark triangle	$s + s^2$	$2s$
Dark square	$2s$	$s + s^2$
Light triangle	$2s$	$s + s^2$
Light square	$s + s^2$	$2s$

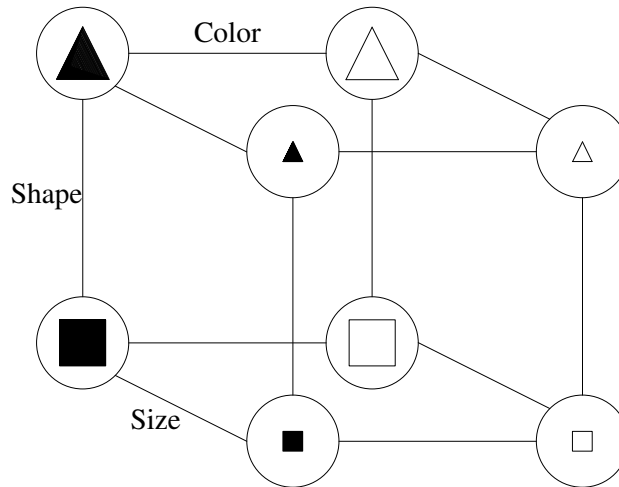


Figure 0.13: Representation of stimuli with shape in the horizontal plane and color in one of the vertical planes. Adapted from Shepard.^[400]

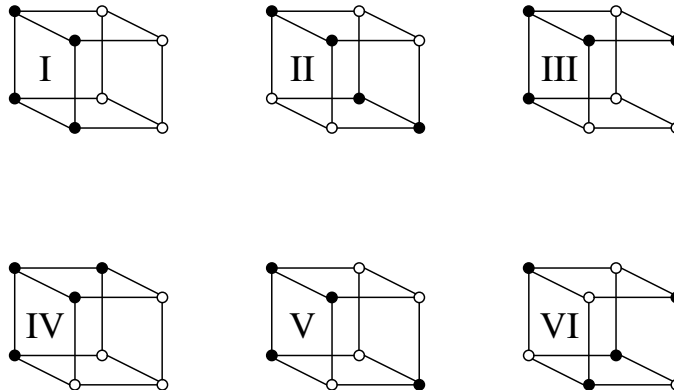


Figure 0.14: One of the six unique configurations (i.e., it is not possible to rotate one configuration into another within the set of six) of selecting four times from eight possibilities. Adapted from Shepard.^[400]

If a subject is shown a stimulus that belongs in category A, the expected probability of them assigning it to that category is:

$$\frac{1 + s^2}{(2s) + (1 + s^2)} \Rightarrow \frac{1 + s^2}{(1 + s)^2} \quad (0.20)$$

For all values of s between 0 and 1 (but not those two values), the probability of a subject assigning a stimulus to the correct category is always less than for the category defined previously, in this case.

In the actual study performed by Shepard, Hovland, and Jenkins,^[400] stimuli that had three attributes, color/size/shape, were used. If there are two possible values for each of these attributes, there are eight possible stimuli (see Figure 0.13).

Each category was assigned four different members. There are 70 different ways of taking four things from a choice of eight ($8!/(4!4!)$), creating 70 possible categories. However, many of these 70 different categories share a common pattern; for instance, all having one attribute, like all black or all triangles. If this symmetry is taken into account, there are only six different kinds of categories. One such selection of six

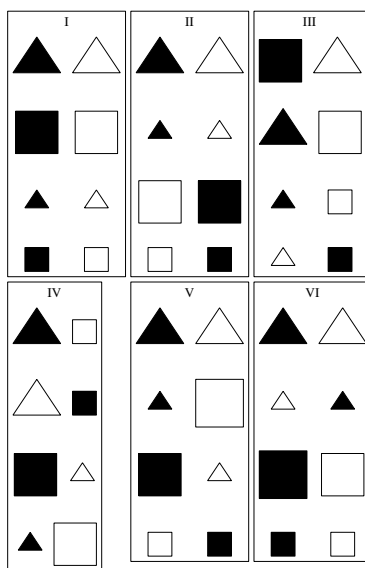


Figure 0.15: Example list of categories. Adapted from Shepard.^[400]

categories is shown in Figure 0.14, the black circles denoting the selected attributes.

Having created these six categories, Shepard et al. trained and measured the performance of subjects in assigning presented stimuli (one of the list of 70 possible combinations of four things— Figure 0.15) to one of them.

Estes^[116] found a reasonable level of agreement between the error rates reported by Shepard et al. and the rates predicted by the similarity to category equations. There is also a connection between categorization performance and Boolean complexity; this is discussed elsewhere.

A series of studies by Feldman^[123] was able to show a correlation between the difficulty subjects had answering the Shepard classification problems and their boolean complexity (i.e., the length of the shortest logically equivalent propositional formula).

13.3 Cultural background and use of information

The attributes used to organize information (e.g., categorize objects) has been found to vary across cultures^[326] and between experts and non-experts. The following studies illustrate how different groups of people agree or differ in their categorization behavior (a cultural difference in the naming of objects is discussed elsewhere):

- A study by Bailenson, Shum, and Coley^[29] asked US bird experts (average of 22.4 years bird watching), US undergraduates, and ordinary Itzaj (Maya Amerindians people from Guatemala) to sort two sets (of US and Maya) of 104 bird species into categories. The results found that the categorization choices made by the three groups of subjects were internally consistent within each group. The correlation between the taxonomies, created by the categories, and a published scientific taxonomy of US experts (0.60 US birds, 0.70 Maya birds), Itzaj (0.45, 0.61), and nonexperts (0.38, 0.34). The US experts correlated highly with the scientific taxonomy for both sets of birds, the Itzaj only correlated highly for Maya birds, and the nonexperts had a low correlation for either set of birds. The reasons given for the Maya choices varied between the expert groups; US experts were based on a scientific taxonomy, Itzaj were based on ecological justifications (the birds relationship with its environment). Cultural differences were found in that, for instance, US subjects were more likely to generalise from songbirds, while the Itzaj were more likely to generalize from perceptually striking birds.
- A study by Proffitt, Coley, and Medin^[368] told three kinds of tree experts (landscape gardeners, parks

maintenance workers, scientists researching trees) about a new disease that affected two kinds of tree (e.g., Horsechestnut and Ohio buckeye). Subjects were then asked what other trees they thought might also be affected by this disease. The results showed differences between kinds of experts in the kinds of justifications given for the answers. For instance, landscapers and maintenance workers used more causal/ecological explanations (tree distribution, mechanism of disease transmission, resistance, and susceptibility) and fewer similarity-based justifications (species diversity and family size). For taxonomists this pattern was reversed.

14 Decision making

Writing source code is not a repetitive process. Developers have to think about what they are going to write, which means they have to make decisions. Achieving the stated intent of these coding guidelines (minimizing the cost of ownership source code) requires that they be included in this, developer, decision-making process.

o coding
guidelines
introduction

There has been a great deal of research into how and why people make decisions in various contexts. For instance, consumer research trying to predict how a shopper will decide among packets of soap powder on a supermarket shelf. While the items being compared and their attributes vary (e.g., which soap will wash the whitest, should an **if** statement or a **switch** statement be used; which computer is best), the same underlying set of mechanisms appear to be used, by people, in making decisions.

The discussion in this section has been strongly influenced by *The Adaptive Decision Maker* by Payne, Bettman, and Johnson.^[340] The model of human decision making proposed by Payne et al. is based on the idea that people balance the predicted cognitive effort required to use a particular decision-making strategy against the likely accuracy achieved by that decision-making strategy. The book lists the following major assumptions:

- *Decision strategies are sequences of mental operations that can be usefully represented as productions of the form IF (condition 1, . . . , condition n) THEN (action 1, . . . , action m).*
- *The cognitive effort needed to reach a decision using a particular strategy is a function of the number and type of operators (productions) used by that strategy, with the relative effort levels of various strategies contingent on task environments.*
- *Different strategies are characterized by different levels of accuracy, with the relative accuracy levels of various strategies contingent on task environments.*
- *As a result of prior experiences and training, a decision maker is assumed to have more than one strategy (sequence of operations) available to solve a decision problem of any complexity.*
- *Individuals decide how to decide primarily by considering both the cognitive effort and the accuracy of the various strategies.*
- *Additional considerations, such as the need to justify a decision to others or the need to minimize the conflict inherent in a decision problem, may also impact strategy selection.*
- *The decision of how to decide is sometimes a conscious choice and sometimes a learned contingency among elements of the task and the relative effort and accuracy of decision strategies.*
- *Strategy selection is generally adaptive and intelligent, if not optimal.*

Payne^[340]

14.1 Decision-making strategies

Before a decision can be made it is necessary to select a decision-making strategy. For instance, a developer who is given an hour to write a program knows there is insufficient time for making complicated trade-offs among alternatives. When a choice needs to be made, the likely decision-making strategy adopted would be to compare the values of a single attribute, the estimated time required to write the code (a decision-making strategy based on looking at the characteristics of a single attribute is known as the lexicographic heuristic).

o lexicographic
heuristic
decision making

Researchers have found that people use a number of different decision-making strategies. In this section we discuss some of these strategies and the circumstances under which people might apply them. The

list of strategies discussed in the following subsections is not exhaustive, but it does cover many of the decision-making strategies used when writing software.

The strategies differ in several ways. For instance, some make trade-offs among the attributes of the alternatives (making it possible for an alternative with several good attributes to be selected instead of the alternative whose only worthwhile attribute is excellent), while others make no such trade-offs. From the human perspective, they also differ in the amount of information that needs to be obtained and the amount of (brain) processing needed to make a decision. A theoretical analysis of the cost of decision making is given by Shugan.^[405]

14.1.1 *The weighted additive rule*

weighted additive rule

The weighted additive rule requires the greatest amount of effort, but delivers the most accurate result. It also requires that any conflicts among different attributes be confronted. Confronting conflict is something, as we shall see later, that people do not like doing. This rule consists of the following steps:

1. Build a list of attributes for each alternative.
2. Assign a value to each of these attributes.
3. Assign a weight to each of these attributes (these weights could, for instance, reflect the relative importance of that attribute to the person making the decision, or the probability of that attribute occurring).
4. For each alternative, sum the product of each of its attributes' value and weight.
5. Select the alternative with the highest sum.

An example, where this rule might be applied, is in deciding whether an equality test against zero should be made before the division of two numbers inside a loop. Attributes might include performance and reliability. If a comparison against zero is made the performance will be decreased by some amount. This disadvantage will be given a high or low weight depending on whether the loop is time-critical or not. The advantage is that reliability will be increased because the possibility of a divide by zero can be avoided. If a comparison against zero is not made, there is no performance penalty, but the reliability could be affected (it is necessary to take into account the probability of the second operand to the divide being zero).

14.1.2 *The equal weight heuristic*

The equal weight heuristic is a simplification of the weighted additive rule in that it assigns the same weight to every attribute. This heuristic might be applied when accurate information on the importance of each attribute is not available, or when a decision to use equal weights has been made.

14.1.3 *The frequency of good and bad features heuristic*

People do not always have an evaluation function for obtaining the value of an attribute. A simple estimate in terms of good/bad is sometimes all that is calculated (looking at things in black and white). By reducing the range of attribute values, this heuristic is a simplification of the equal weight heuristic, which in turn is a simplification of the weighted additive rule. This rule consists of the following steps:

1. List the good and bad attributes of every alternative.
2. Calculate the sum of each attributes good and the sum of its bad attributes.
3. Select the alternative with the highest count of either good or bad attributes, or some combination of the two.

A coding context, where a good/bad selection might be applicable, occurs in choosing the type of an object. If the object needs to hold a fractional part, it is tempting to use a floating type rather than an integer type (perhaps using some scaling factor to enable the fractional part to be represented). Drawing up a list of good and bad attributes ought to be relatively straight-forward; balancing them, to select a final type, might be a little more contentious

14.1.4 The majority of confirming dimensions heuristic

While people may not be able to explicitly state an evaluation function that provides a numerical measure of an attribute, they can often give a yes/no answer to the question: *Is the value of attribute X greater (or less) for alternative A compared to alternative B?*. This enables them to determine which alternative has the most (or least) of each attribute. This rule consists of the following steps:

1. Select a pair of alternatives.
2. Compare each matching attribute in the two alternatives.
3. Select the alternative that has the greater number of winning attributes.
4. Pair the winning alternative with an unpaired alternative and repeat the compare/select steps.
5. Once all alternatives have been compared at least once, the final winning alternative is selected.

In many coding situations there are often only two viable alternatives. Pairwise comparison of their attributes could be relatively easy to perform. For instance, when deciding whether to use a sequence of **if** statements or a **switch** statement, possible comparison attributes include efficiency of execution, readability, ease of changeability (adding new cases, deleting, or merging existing ones).

14.1.5 The satisficing heuristic

The result of the satisficing heuristic depends on the order in which alternatives are checked and often does not check all alternatives. Such a decision strategy, when described in this way, sounds unusual, but it is simple to perform. This rule consists of the following steps:

satisficing
heuristic
decision making

1. Assign a cutoff, or aspirational, level that must be met by each attribute.
2. Perform the following for each alternative:
 - Check each of its attributes against the cutoff level, rejecting the alternative if the attribute is below the cutoff.
 - If there are no attributes below the cutoff value, accept this alternative.
3. If no alternative is accepted, revise the cutoff levels associated the attributes and repeat the previous step.

An example of the satisficing heuristic might be seen when selecting a library function to return some information to a program. The list of attributes might include the amount of information returned and the format it is returned in (relative to the format it is required to be in). Once a library function meeting the developer's minimum aspirational level has been found, additional effort is not usually invested in finding a better alternative.

14.1.6 The lexicographic heuristic

The lexicographic heuristic has a low effort cost, but it might not be very accurate. It can also be intransitive; with X preferred to Y, Y preferred to Z, and Z preferred to X. This rule consists of the following steps:

lexicographic
heuristic
decision making

1. Determine the most important attribute of all the alternatives.
2. Find the alternative that has the best value for the selected most important attribute.
3. If two or more alternatives have the same value, select the next most important attribute and repeat the previous step using the set of alternatives whose attribute values tied.
4. The result is the alternative having the best value on the final, most important, attribute selected.

An example of the intransitivity that can occur, when using this heuristic, might be seen when writing software for embedded applications. Here the code has to fit within storage units that occur in fixed-size increments (e.g., 8 K chips). It may be possible to increase the speed of execution of an application by writing code for specific special cases; or have generalized code that is more compact, but slower. We might have the following, commonly seen, alternatives (see Table 0.11).

Table 0.11: Storage/Execution performance alternatives.

Alternative	Storage Needed	Speed of Execution
X	7 K	Low
Y	15 K	High
Z	10 K	Medium

Based on storage needed, X is preferred to Y. But because storage comes in 8 K increments there is no preference, based on this attribute, between Y and Z; however, Y is preferred to Z based on speed of execution. Based on speed of execution Z is preferred to X.

14.1.6.1 The elimination-by-aspects heuristic

The elimination-by-aspects heuristic uses cutoff levels, but it differs from the satisficing heuristic in that alternatives are eliminated because their attributes fall below these levels. This rule consists of the following steps:

1. The attributes for all alternatives are ordered (this ordering might be based on some weighting scheme).
2. For each attribute in turn, starting with the most important, until one alternative remains:
 - Select a cutoff value for that attribute.
 - Eliminate all alternatives whose value for that attribute is below the cutoff.
3. Select the alternative that remains.

This heuristic is often used when there are resource limitations, for instance, deadlines to meet, performance levels to achieve, or storage capacities to fit within.

14.1.7 The habitual heuristic

The habitual heuristic looks for a match of the current situation against past situations, it does not contain any evaluation function (although there are related heuristics that evaluate the outcome of previous decisions). This rule consists of the step:

1. select the alternative chosen last time for that situation.

Your author’s unsubstantiated claim is that this is the primary decision-making strategy used by software developers.

Sticking with a winning solution suggests one of two situations:

1. So little is known that once a winning solution is found, it is better to stick with it than to pay the cost (time and the possibility of failure) of looking for a better solution that might not exist.
2. The developer has extensively analyzed the situation and knows the best solution.

Coding decisions are not usually of critical importance. There are many solutions that will do a satisfactory job. It may also be very difficult to measure the effectiveness of any decision, because there is a significant delay between the decision being made and being able to measure its effect. In many cases, it is almost

impossible to separate out the effect of one decision from the effects of all the other decisions made (there may be a few large coding decisions, but the majority are small ones).

A study by Klein^[225] describes how fireground commanders use their experience to size-up a situation very rapidly. Orders are given to the firefighters under their command without any apparent decisions being made (in their interviews they even found a fireground commander who claimed that neither he, nor other commanders, ever made decisions; they knew what to do). Klein calls this strategy *recognition-primed decision making*.

recognition-
primed
decision making

14.2 Selecting a strategy

Although researchers have uncovered several decision-making strategies that people use, their existence does not imply that people will make use of all of them. The strategies available to individuals can vary depending on their education, training, and experience. A distinction also needs to be made between a person's knowledge of a strategy (through education and training) and their ability to successfully apply it (perhaps based on experience).

The task itself (that creates the need for a decision to be made) can affect the strategy used. These task effects include task complexity, the response mode (how the answer needs to be given), how the information is displayed, and context. The following subsections briefly outline these effects.

14.2.1 Task complexity

In general the more complex the decision, the more people will tend to use simplifying heuristics. The following factors influence complexity:

task complexity
decision making

- *Number of alternatives.* As the number of alternatives that need to be considered grows, there are shifts in the decision-making strategy used.
- *Number of attributes.* Increasing the number of attributes increases the confidence of people's judgments, but it also increases their variability. The evidence for changes in the quality of decision making, as the number of attributes increases, is less clear-cut. Some studies show a decrease in quality; it has been suggested that people become overloaded with information. There is also the problem of deciding what constitutes a high-quality decision.
- *Time pressure.* People have been found to respond to time pressure in one of several ways. Some respond by accelerating their processing of information, others respond by reducing the amount of information they process (by filtering the less important information, or by concentrating on certain kinds of information such as negative information), while others respond by reducing the range of ideas and actions considered.

14.2.2 Response mode

There are several different response modes. For instance, a choice response mode frames the alternatives in terms of different choices; a matching response mode presents a list of questions and answers and the decision maker has to provide a matching answer to a question; a bidding response mode requires a value to be given for buying or selling some object. There are also other response modes, that are not listed here.

The choice of response mode, in some cases, has been shown to significantly influence the preferred alternatives. In extreme cases, making a decision may result in X being preferred to Y, while the mathematically equivalent decision, presented using a different response mode, can result in Y being preferred to X. For instance, in gambling situations it has been found that people will prefer X to Y when asked to select between two gambles (where X has a higher probability of winning, but with lower amounts), but when asked to bid on gambles they prefer Y to X (with Y representing a lower probability of winning a larger amount).

Such behavior breaks what was once claimed to be a fundamental principle of rational decision theory, *procedure invariance*. The idea behind this principle was that people had an invariant (relatively) set of internal preferences that were used to make decisions. These experiments showed that sometimes preferences

are constructed on the fly. Observed preferences are likely to take a person's internal preferences and the heuristics used to construct the answer into account.

Code maintenance is one situation where the task can have a large impact on how the answer is selected. When small changes are made to existing code, many developers tend to operate in a matching mode, choosing constructs similar, if not identical, to the ones in the immediately surrounding lines of code. If writing the same code from scratch, there is nothing to match, another response mode will necessarily need to be used in deciding what constructs to use.

A lot of the theoretical discussion on the reasons for these response mode effects has involved distinguishing between judgment and choice. People can behave differently, depending on whether they are asked to make a judgment or a choice. When writing code, the difference between judgment and choice is not always clear-cut. Developers may believe they are making a choice between two constructs when in fact they have already made a judgment that has reduced the number of alternatives to choose between.

Writing code is open-ended in the sense that theoretically there are an infinite number of different ways of implementing what needs to be done. Only half a dozen of these might be considered sensible ways of implementing some given functionality, with perhaps one or two being commonly used. Developers often limit the number of alternatives under consideration because of what they perceive to be overriding external factors, such as preferring an inline solution rather than calling a library function because of alleged quality problems with that library. One possibility is that decision making during coding be considered as a two-stage process, using judgment to select the alternatives, from which one is chosen.

14.2.3 Information display

Studies have shown that how information, used in making a decision, is displayed can influence the choice of a decision-making strategy.^[392] These issues include: only using the information that is visible (the concreteness principle), the difference between available information and processable information (displaying the price of one brand of soap in dollars per ounce, while another brand displays francs per kilogram), the completeness of the information (people seem to weigh common attributes more heavily than unique ones, perhaps because of the cognitive ease of comparison), and the format of the information (e.g., digits or words for numeric values).

What kind of information is on display when code is being written? A screen's worth of existing code is visible on the display in front of the developer. There may be some notes to the side of the display. All other information that is used exists in the developer's head.

Existing code is the result of past decisions made by the developer; it may also be modified by future decisions that need to be made (because of a need to modify the behavior of this existing code). For instance, the case in which another conditional statement needs to be added within a deeply nested series of conditionals. The information display (layout) of the existing code can affect the developer's decision about how the code is to be modified (a function, or macro, might be created instead of simply inserting the new conditional). Here the information display itself is an attribute of the decision making (code wrapping, at the end of a line, is an attribute that has a yes/no answer).

14.2.4 Agenda effects

The agenda effect occurs when the order in which alternatives are considered influences the final answer. For instance, take alternatives X, Y, and Z and group them into some form of hierarchy before performing a selection. When asked to choose between the pair [X, Y] and Z (followed by a choice between X and Y if that pair is chosen) and asked to choose between the pair [X, Z] and Y (again followed by another choice if that pair is chosen), an agenda effect would occur if the two final answers were different.

An example of the agenda effect is the following. When writing coding, it is sometimes necessary to decide between writing in line code, using a macro, or using a function. These three alternatives can be grouped into a natural hierarchy depending on the requirements. If efficiency is a primary concern, the first decision may be between [in line, macro] and function, followed by a decision between in line and macro (if that pair is chosen). If we are more interested in having some degree of abstraction, the first decision is likely to be between [macro, function] and in line (see Figure 0.16).

agenda effects
decision making

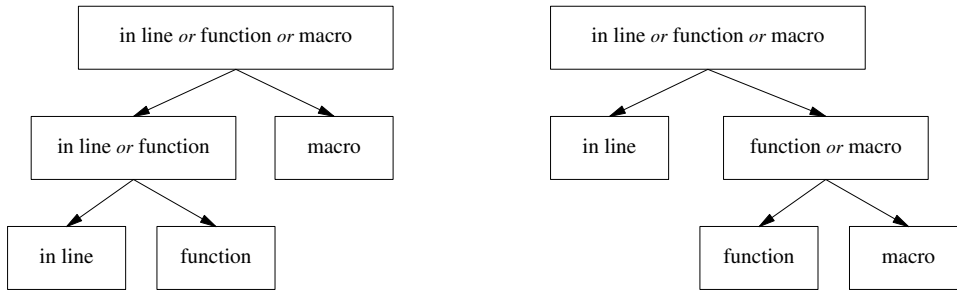


Figure 0.16: Possible decision paths when making pair-wise comparisons on whether to use an inline code, a function, or a macro; for two different pair-wise associations.

In the efficiency case, if performance is important in the context of the decision, [in line, macro] is likely to be selected in preference to function. Once this initial choice has been made other attributes can be considered (since both alternatives have the same efficiency). We can now decide whether abstraction is considered important enough to select macro over in line.

If the initial choice had been between [macro, function] and in line, the importance of efficiency would have resulted in in line being chosen (when paired with function, macro appears less efficient by association).

14.2.5 Matching and choosing

When asked to make a decision based on *matching*, a person is required to specify the value of some variable such that two alternatives are considered to be equivalent. For instance, how much time should be spent testing 200 lines of code to make it as reliable as the 500 lines of code that has had 10 hours of testing invested in it? When asked to make a decision based on *choice*, a person is presented with a set of alternatives and is required to specify one of them.

A study by Tversky, Sattath, and Slovic^[465] investigated the *prominence hypothesis*. This proposes that when asked to make a decision based on choice, people tend to use the prominent attributes of the options presented (adjusting unweighted intervals being preferred for matching options). Their study suggested that there were differences between the mechanisms used to make decisions for matching and choosing.

14.3 The developer as decision maker

The writing of source code would seem to require developers to make a very large number of decisions. However, experience shows that developers do not appear to be consciously making many decisions concerning what code to write. Most decisions being made involve issues related to the mapping from the application domain, choosing algorithms, and general organizational issues (i.e., where functions or objects should be defined).

Many of the coding-level decisions that need to be made occur again and again. Within a year or so, in full-time software development, sufficient experience has usually been gained for many decisions to be reduced to matching situations against those previously seen, and selecting the corresponding solution. For instance, the decision to use a series of **if** statements or a **switch** statement might require the pattern *same variable tested against integer constant and more than two tests are made to be true before a switch statement is used*. This is what Klein^[225] calls recognition-primed decision making. This code writing methodology works because there is rarely a need to select the optimum alternative from those available.

0 recognition-primed decision making

Some decisions occur to the developer as code is being written. For instance, a developer may notice that the same sequence of statements, currently being written, was written earlier in a different part of the source (or perhaps it will occur to the developer that the same sequence of statements is likely to be needed in code that is yet to be written). At this point the developer has to make a decision about making a decision (metacognition). Should the decision about whether to create a function be put off until the current work item is completed, or should the developer stop what they are currently doing to make a decision on whether to

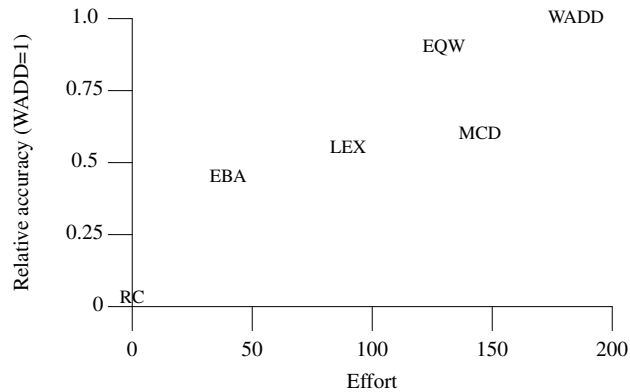


Figure 0.17: Effort and accuracy levels for various decision-making strategies; EBA (Elimination-by-aspects heuristic), EQW (equal weight heuristic), LEX (lexicographic heuristic), MCD (majority of confirming dimensions heuristic), RC (Random choice), and WADD (weighted additive rule). Adapted from Payne.^[340]

turn the statement sequence into a function definition? Remembering work items and metacognitive decision processes are handled by a developer's attention. The subject of *attention* is discussed elsewhere.

Just because developers are not making frequent, conscious decisions does not mean that their choices are consistent and repeatable (they will always make the same decision). There are a number of both internal and external factors that may affect the decisions made. Researchers have uncovered a wide range of issues, a few of which are discussed in the following subsections.

14.3.1 Cognitive effort vs. accuracy

People like to make accurate decisions with the minimum of effort. In practice, selecting a decision-making strategy requires trading accuracy against effort (or to be exact, expected effort making the decision; the actual effort required can only be known after the decision has been made).

The fact that people do make effort/accuracy trade-offs is shown by the results from a wide range of studies (this issue is also discussed elsewhere, and Payne et al.^[340] discuss this topic in detail). See Figure 0.17 for a comparison.

The extent to which any significant cognitive effort is expended in decision making while writing code is open to debate. A developer may be expending a lot of effort on thinking, but this could be related to problem solving, algorithmic, or design issues.

One way of performing an activity that is not much talked about, is *flow*— performing an activity without any conscious effort— often giving pleasure to the performer. A best-selling book on the subject of flow^[90] is subtitled “The psychology of optimal experience”, something that artistic performers often talk about. Developers sometimes talk of *going with the flow* or *just letting the writing flow* when writing code; something writers working in any medium might appreciate. However, it is your author's experience that this method of working often occurs when deadlines approach and developers are faced with writing a lot of code quickly. Code written using *flow* is often very much like a river; it has a start and an ending, but between those points it follows the path of least resistance, and at any point readers rarely have any idea of where it has been or where it is going. While works of fiction may gain from being written in this way, the source code addressed by this book is not intended to be read for enjoyment. While developers may enjoy spending time solving mysteries, their employers do not want to pay them to have to do so.

Code written using *flow* is not recommended, and is not discussed further here. The use of intuition is discussed elsewhere.

14.3.2 Which attributes are considered important?

Developers tend to consider mainly technical attributes when making decisions. Economic attributes are often ignored, or considered unimportant. No discussion about attributes would be complete without mentioning

fun. Developers have gotten used to the idea that they can enjoy themselves at work, doing *fun* things. Alternatives that have a negative value for the fun attribute, and a large positive value for the time to carry out attribute are often quickly eliminated.

The influence of developer enjoyment on decision making, can be seen in many developers' preference for writing code, rather than calling a library function. On a larger scale, the often-heard developer recommendation for rewriting a program, rather than reengineering an existing one, is motivated more by the expected pleasure of writing code than the economics (and frustration) of reengineering.

One reason for the lack of consideration of economic factors is that many developers have no training, or experience in this area. Providing training is one way of introducing an economic element into the attributes used by developers in their decision making.

14.3.3 Emotional factors

Many people do not like being in a state of conflict and try to avoid it. Making a decision can create conflict, by requiring one attribute to be traded off against another. For instance, having to decide whether it is more important for a piece of code to execute quickly or reliably. It has been argued that people will avoid strategies that involve difficult, emotional, value trade-offs.

developer
emotional factors
o weighted
additive rule

Emotional factors relating to source code need not be limited to internal, private developer decision making. During the development of an application involving more than one developer, particular parts of the source are often considered to be *owned* by an individual developer. A developer asked to work on another developers source code, perhaps because that person is away, will sometimes feel the need to adopt the *style* of that developer, making changes to the code in a way that is thought to be acceptable to the absent developer. Another approach is to ensure that the changes stand out from the *owner's* code. On the owning developer's return, the way in which changes were made is explained. Because they stand out, developers can easily see what changes were made to *their* code and decide what to do about them.

People do not like to be seen to make mistakes. It has been proposed^[107] that people have difficulty using a decision-making strategy, that makes it explicit that there is some amount of error in the selected alternative. This behavior occurs even when it can be shown that the strategy would lead to better, on average, solutions than the other strategies available.

14.3.4 Overconfidence

A person is overconfident when their belief in a proposition is greater than is warranted by the information available to them. It has been argued that overconfidence is a useful attribute that has been selected for by evolution. Individuals who overestimates their ability are more likely to undertake activities they would not otherwise have been willing to do. Taylor and Brown^[443] argue that a theory of mental health defined in terms of contact with reality does not itself have contact with reality: "Rather, the mentally healthy person appears to have the enviable capacity to distort reality in a direction that enhances self-esteem, maintains beliefs in personal efficacy, and promotes an optimistic view of the future."

overconfidence

Numerous studies have shown that most people are overconfident about their own abilities compared with others. People can be overconfident in their ability for several reasons: confirmation bias can lead to available information being incorrectly interpreted; a person's inexpert calibration (the degree of correlation between confidence and performance) of their own abilities is another reason. A recent study^[224] has also highlighted the importance of the how, what, and whom of questioning in overconfidence studies. In some cases, it has been shown to be possible to make overconfidence disappear, depending on how the question is asked, or on what question is asked. Some results also show that there are consistent individual differences in the degree of overconfidence.

o confirmation
bias

ignorance more frequently begets confidence than does knowledge

Charles Darwin, In
The descent of man,
1871, p. 3

A study by Glenberg and Epstein^[146] showed the danger of a little knowledge. They asked students, who were studying either physics or music, to read a paragraph illustrating some central principle (of physics or music). Subjects were asked to rate their confidence in being able to accurately answer a question about

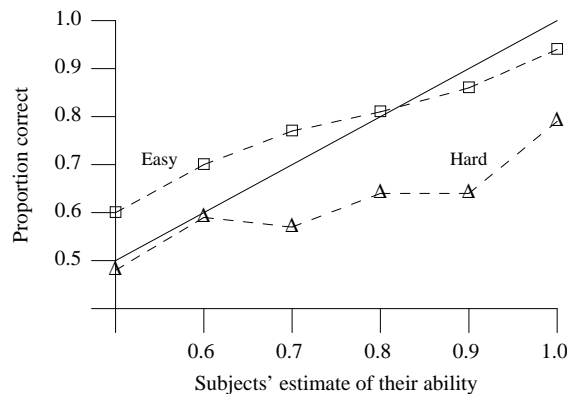


Figure 0.18: Subjects' estimate of their ability (bottom scale) to correctly answer a question and actual performance in answering on the left scale. The responses of a person with perfect self-knowledge is given by the solid line. Adapted from Lichtenstein.^[264]

the text. They were then presented with a statement drawing some conclusion about the text (it was either true or false), which they had to answer. They then had to rate their confidence that they had answered the question correctly. This process was repeated for a second statement, which differed from the first in having the opposite true/false status.

The results showed that the more physics or music courses a subject had taken, the more confident they were about their own abilities. However, a subject's greater confidence in being able to correctly answer a question, before seeing it, was not matched by a greater ability to provide the correct answer. In fact as subjects' confidence increased, the accuracy of the calibration of their own ability went down. Once they had seen the question, and answered it, subjects were able to accurately calibrate their performance.

Subjects did not learn from their previous performances (in answering questions). They could have used information on the discrepancy between their confidence levels before/after seeing previous questions to improve the accuracy of their confidence estimates on subsequent questions.

The conclusion drawn by Glenberg and Epstein was that subjects' overconfidence judgments were based on self-classification as an expert, within a domain, not the degree to which they comprehended the text.

A study by Lichtenstein and Fishhoff^[264] discovered a different kind of overconfidence effect. As the difficulty of a task increased, the accuracy of people's estimates of their own ability to perform the task decreased. In this study subjects were asked general knowledge questions, with the questions divided into two groups, hard and easy. The results in Figure 0.18 show that subjects' overestimated their ability (bottom scale) to correctly answer (actual performance, left scale) hard questions. On the other hand, they underestimated their ability to answer easy questions. The responses of a person with perfect self-knowledge are given by the solid line.

These, and subsequent results, show that the skills and knowledge that constitute competence in a particular domain are the same skills needed to evaluate one's (and other people's) competence in that domain. People who do not have these skills and knowledge lack metacognition (the name given by cognitive psychologists to the ability of a person to accurately judge how well they are performing). In other words, the knowledge that underlies the ability to produce correct judgment is the same knowledge that underlies the ability to recognize correct judgment.

Some very worrying results, about what overconfident people will do, were obtained in a study performed by Arkes, Dawes, and Christensen.^[19] This study found that subjects used a formula that calculated the best decision in a probabilistic context (provided to them as part of the experiment) less when incentives were provided or the subjects thought they had domain expertise. This behavior even continued when the subjects were given feedback on the accuracy of their own decisions. The explanation, given by Arkes et al., was that when incentives were provided, people changed decision-making strategies in an attempt to beat the odds. Langer^[245] calls this behavior *the illusion of control*.

Developers overconfidence and their aversion to explicit errors can sometimes be seen in the handling of floating-point calculations. A significant amount of mathematical work has been devoted to discovering the bounds on the errors for various numerical algorithms. Sometimes it has been proved that the error in the result of a particular algorithm is the minimum error attainable (there is no algorithm whose result has less error). This does not seem to prevent some developers from believing that they can design a more accurate algorithm. Phrases, such as *mean error* and *average error*, in the presentation of an algorithm's error analysis do not help. An overconfident developer could take this as a hint that it is possible to do better for the conditions that prevail in his (or her) application (and not having an error analysis does not disprove it is not better).

14.4 The impact of guideline recommendations on decision making

A set of guidelines can be more than a list of recommendations that provide a precomputed decision matrix. A guidelines document can provide background information. Before making any recommendations, the author(s) of a guidelines document need to consider the construct in detail. A good set of guidelines will document these considerations. This documentation provides a knowledge base of the alternatives that might be considered, and a list of the attributes that need to be taken into account. Ideally, precomputed values and weights for each attribute would also be provided. At the time of this writing your author only has a vague idea about how these values and weights might be computed, and does not have the raw data needed to compute them.

A set of guideline recommendations can act as a lightning rod for decisions that contain an emotional dimension. Adhering to coding guidelines being the justification for the decision that needs to be made. Having to justify decisions can affect the decision-making strategy used. If developers are expected to adhere to a set of guidelines, the decisions they make could vary depending on whether the code they write is independently checked (during code review, or with a static analysis tool).

14.5 Management's impact on developers' decision making

Although lip service is generally paid to the idea that coding guidelines are beneficial, all developers seem to have heard of a case where having to follow guidelines has been counterproductive. In practice, when first introduced, guidelines are often judged by both the amount of additional work they create for developers and the number of faults they immediately help locate. While an automated tool may uncover faults in existing code, this is not the primary intended purpose of using these coding guidelines. The cost of adhering to guidelines in the present is paid by developers; the benefit is reaped in the future by the owners of the software. Unless management successfully deals with this cost/benefit situation, developers could decide it is not worth their while to adhere to guideline recommendations.

What factors, controlled by management, have an effect on developers' decision making? The following subsections discuss some of them.

14.5.1 Effects of incentives

Some deadlines are sufficiently important that developers are offered incentives to meet them. Studies, on use of incentives, show that their effect seems to be to make people work harder, not necessarily smarter.

Increased effort is thought to lead to improved results. Research by Paese and Sniezek^[333] found that increased effort led to increased confidence in the result, but without there being any associated increase in decision accuracy.

Before incentives can lead to a change of decision-making strategies, several conditions need to be met:

- The developer must believe that a more accurate strategy is required. Feedback on the accuracy of decisions is the first step in highlighting the need for a different strategy,^[171] but it need not be sufficient to cause a change of strategy.
- A better strategy must be available. The information needed to be able to use alternative strategies may not be available (for instance, a list of attribute values and weights for a weighted average strategy).
- The developer must believe that they are capable of performing the strategy.

14.5.2 Effects of time pressure

Research by Payne, Bettman, and Johnson,^[340] and others, has shown that there is a hierarchy of responses for how people deal with time pressure:

1. They work faster.
2. If that fails, they may focus on a subset of the issues.
3. If that fails, they may change strategies (e.g., from alternative based to attribute based).

If the time pressure is on delivering a finished program, and testing has uncovered a fault that requires changes to the code, then the weighting assigned to attributes is likely to be different than during initial development. For instance, the risk of a particular code change impacting other parts of the program is likely to be a highly weighted attribute, while maintainability issues are usually given a lower weighting as deadlines approach.

14.5.3 Effects of decision importance

Studies investigating at how people select decision-making strategies have found that increasing the benefit for making a correct decision, or having to make a decision that is irreversible, influences how rigorously a strategy is applied, not which strategy is applied.^[37]

The same coding construct can have a different perceived importance in different contexts. For instance, defining an object at file scope is often considered to be a more important decision than defining one in block scope. The file scope declaration has more future consequences than the one in block scope.

An irreversible decision might be one that selects the parameter ordering in the declaration of a library function. Once other developers have included calls to this function in their code, it can be almost impossible (high cost/low benefit) to change the parameter ordering.

14.5.4 Effects of training

A developer's training in software development is often done using examples. Sample programs are used to demonstrate the solutions to small problems. As well as learning how different constructs behave, and how they can be joined together to create programs, developers also learn what attributes are considered to be important in source code. They learn the implicit information that is not written down in the text books. Sources of implicit learning include the following:

- *The translator used for writing class exercises.* All translators have their idiosyncrasies and beginners are not sophisticated enough to distinguish these from truly generic behavior. A developer's first translator usually colors his view of writing code for several years.
- *Personal experiences during the first few months of training.* There are usually several different alternatives for performing some operation. A bad experience (perhaps being unable to get a program that used a block scope array to work, but when the array was moved to file scope the program worked) with some construct can lead to a belief that use of that construct was problem-prone and to be avoided (all array objects being declared, by that developer, at file scope and never in block scope).
- *Instructor biases.* The person teaching a class and marking submitted solutions will impart their own views on what attributes are important. Efficiency of execution is an attribute that is often considered to be important. Its actual importance, in most cases, has declined from being crucial 50 years ago to being almost a nonissue today. There is also the technical interest factor in trying to write code as efficiently as possible. A related attribute is program size. Praise is more often given for short programs, rather than longer ones. There are applications where the size of the code is important, but generally time spent writing the shortest program is wasted (and may even be more difficult to comprehend than a longer program).
- *Consideration for other developers.* Developers are rarely given practical training on how to read code, or how to write code that can easily be read by others. Developers generally believe that any difficulty others experience in comprehending their code is not caused by how they wrote it.

- *Preexisting behavior.* Developers bring their existing beliefs and modes of working to writing C source. These can range from behavior that is not software-specific, such as the inability to ignore sunk costs (i.e., wanting to modify an existing piece of code, they wrote earlier, rather than throw it away and starting again; although this does not seem to apply to throwing away code written by other people), to the use of the idioms of another language when writing in C.
- *Technically based.* Most existing education and training in software development tends to be based on purely technical issues. Economic issues are not usually raised formally, although informally time-to-completion is recognized as an important issue.

Unfortunately, once most developers have learned an initial set of attribute values and weightings for source code constructs, there is usually a long period of time before any subsequent major tuning or relearning takes place. Developers tend to be too busy applying their knowledge to question many of the underlying assumptions they have picked up along the way.

Based on this background, it is to be expected that many developers will harbor a few *myths* about what constitutes a good coding decision in certain circumstances. These coding guidelines cannot address all coding myths. Where appropriate, coding myths commonly encountered by your author are discussed.

14.5.5 Having to justify decisions

Studies have found that having to justify a decision can affect the choice of decision-making strategy to be used. For instance, Tetlock and Boettger^[448] found that subjects who were accountable for their decisions used a much wider range of information in making judgments. While taking more information into account did not necessarily result in better decisions, it did mean that additional information that was both irrelevant and relevant to the decision was taken into account.

It has been proposed, by Tversky,^[461] that the elimination-by-aspects heuristic is easy to justify. However, while use of this heuristic may make for easier justification, it need not make for more accurate decisions.

A study performed by Simonson^[413] showed that subjects who had difficulty determining which alternative had the greatest utility tended to select the alternative that supported the best overall reasons (for choosing it).

Tetlock^[447] included an accountability factor into decision-making theory. One strategy that handles accountability as well as minimizing cognitive effort is to select the alternative that the perspective audience (i.e., code review members) is thought most likely to select. Not knowing which alternative they are likely to select can lead to a more flexible approach to strategies. The exception occurs when a person has already made the decision; in this case the cognitive effort goes into defending that decision.

During a code review, a developer may have to justify why a particular decision was made. While developers know that time limits will make it very unlikely that they will have to justify every decision, they do not know in advance which decisions will have to be justified. In effect, the developer will feel the need to be able to justify most decisions.

Requiring developers to justify why they have not followed a particular guideline recommendation can be a two-edged sword. Developers can respond by deciding to blindly follow guidelines (the path of least resistance), or they can invest effort in evaluating, and documenting, the different alternatives (not necessarily a good thing since the invested effort may not be warranted by the expected benefits). The extent to which some people will blindly obey authority was chillingly demonstrated in a number of studies by Milgram.^[295]

14.6 Another theory about decision making

The theory that selection of a decision-making strategy is based on trading off cognitive effort and accuracy is not the only theory that has been proposed. Hammond, Hamm, Grassia, and Pearson^[159] proposed that analytic decision making is only one end of a continuum; at the other end is intuition. They performed a study, using highway engineers, involving three tasks. Each task was designed to have specific characteristics (see Table 0.12). One task contained intuition-inducing characteristics, one analysis-inducing, and the third an equal mixture of the two. For the problems studied, intuitive cognition outperformed analytical cognition in terms of the empirical accuracy of the judgments.

justifying
decisions

Table 0.12: Inducement of intuitive cognition and analytic cognition, by task conditions. Adapted from Hammond.^[159]

Task Characteristic	Intuition-Inducing State of Task Characteristic	Analysis-Inducing State of Task Characteristic
Number of cues	Large (>5)	Small
Measurement of cues	Perceptual measurement	Objective reliable measurement
Distribution of cue values	Continuous highly variable distribution	Unknown distribution; cues are dichotomous; values are discrete
Redundancy among cues	High redundancy	Low redundancy
Decomposition of task	Low	High
Degree of certainty in task	Low certainty	High certainty
Relation between cues and criterion	Linear	Nonlinear
Weighting of cues in environmental model	Equal	Unequal
Availability of organizing principle	Unavailable	Available
Display of cues	Simultaneous display	Sequential display
Time period	Brief	Long

One of the conclusions that Hammond et al. drew from these results is that “Experts should increase their awareness of the correspondence between task and cognition”. A task having intuition-inducing characteristics is most likely to be out carried using intuition, and similarly for analysis-inducing characteristics.

Many developers sometimes talk of writing code intuitively. Discussion of intuition and flow of consciousness are often intermixed. The extent to which either intuitive or analytic decision making (if that is how developers operate) is more cost effective, or practical, is beyond this author’s ability to even start to answer. It is mentioned in this book because there is a bona fide theory that uses these concepts and developers sometimes also refer to them.

Intuition can be said to be characterized by rapid data processing, low cognitive control (the consistency with which a judgment policy is applied), and low awareness of processing. Its opposite, analysis, is characterized by slow data processing, high cognitive control, and high awareness of processing.

15 Expertise

People are referred to as being experts, in a particular domain, for several reasons, including:

- Well-established figures, perhaps holding a senior position with an organization heavily involved in that domain.
- Better at performing a task than the average person on the street.
- Better at performing a task than most other people who can also perform that task.
- Self-proclaimed experts, who are willing to accept money from clients who are not willing to take responsibility for proposing what needs to be done.^[197]

Schneider^[393] defines a high-performance skill as one for which (1) more than 100 hours of training are required, (2) substantial numbers of individuals fail to develop proficiency, and (3) the performance of an expert is qualitatively different from that of the novice.

In this section, we are interested in why some people (the experts) are able to give a task performance that is measurably better than a non-expert (who can also perform the task).

There are domains in which those acknowledged as experts do not perform significantly better than those considered to be non-experts.^[59] For instance, in typical cases the performance of medical experts was not much greater than those of doctors after their first year of residency, although much larger differences were seen for difficult cases. Are there domains where it is intrinsically not possible to become significantly better than one’s peers, or are there other factors that can create a large performance difference between expert and non-expert performances? One way to help answer this question is to look at domains where the gap between expert and non-expert performance can be very large.

developer
flow

developer
intuition

expertise

It is a commonly held belief that experts have some innate ability or capacity that enables them to do what they do so well. Research over the last two decades has shown that while innate ability can be a factor in performance (there do appear to be genetic factors associated with some athletic performances), the main factor in acquiring expert performance is time spent in *deliberate practice*.^[112]

Deliberate practice is different from simply performing the task. It requires that people monitor their practice with full concentration and obtain feedback^[171] on what they are doing (often from a professional teacher). It may also involve studying components of the skill in isolation, attempting to improve on particular aspects. The goal of this practice being to improve performance, not to produce a finished product.

Studies of the backgrounds of recognized experts, in many fields, found that the elapsed time between them starting out and carrying out their best work was at least 10 years, often with several hours of deliberate practice every day of the year. For instance, Ericsson, Krampe, and Tesch-Romer^[113] found that, in a study of violinists (a perceptual-motor task), by age 20 those at the top level had practiced for 10,000 hours, those at the next level down 7,500 hours, and those at the lowest level of expertise had practiced for 5,000 hours. They also found similar quantities of practice being needed to attain expert performance levels in purely mental activities (e.g., chess).

People often learn a skill for some purpose (e.g., chess as a social activity, programming to get a job) without the aim of achieving expert performance. Once a certain level of proficiency is achieved, they stop trying to learn and concentrate on using what they have learned (in work, and sport, a distinction is made between training for and performing the activity). During everyday work, the goal is to produce a product or to provide a service. In these situations people need to use well-established methods, not try new (potentially dead-end, or leading to failure) ideas to be certain of success. Time spent on this kind of practice does not lead to any significant improvement in expertise, although people may become very fluent in performing their particular subset of skills.

What of individual aptitudes? In the cases studied by researchers, the effects of aptitude, if there are any, have been found to be completely overshadowed by differences in experience and deliberate practice times. What makes a person willing to spend many hours, every day, studying to achieve expert performance is open to debate. Does an initial aptitude or interest in a subject lead to praise from others (the path to musical and chess expert performance often starts in childhood), which creates the atmosphere for learning, or are other issues involved? IQ does correlate to performance during and immediately after training, but the correlation reduces over the years. The IQ of experts has been found to be higher than the average population at about the level of college students.

In many fields expertise is acquired by memorizing a huge amount of, domain-specific, knowledge and having the ability to solve problems using pattern-based retrieval on this knowledge base. The knowledge is structured in a form suitable for the kind of information retrieval needed for problems in a domain.^[114]

A study by Carlson, Khoo, Yaure, and Schneider^[61] examined changes in problem-solving activity as subjects acquired a skill (trouble shooting problems with a digital circuit). Subjects started knowing nothing, were given training in the task, and then given 347 problems to solve (in 28 individual, two-hour sessions, over a 15-week period). The results showed that subjects made rapid improvements in some areas (and little thereafter), extended practice produced continuing improvement in some of the task components, subjects acquired the ability to perform some secondary tasks in parallel, and transfer of skills to new digital circuits was substantial but less than perfect. Even after 56 hours of practice, the performance of subjects continued to show improvements and had not started to level off. Where are the limits to continued improvements? A study by Crossman^[89] of workers producing cigars showed performance improving according to the power law of practice for the first five years of employment. Thereafter performance improvements slow; factors cited for this slow down include approaching the speed limit of the equipment being used and the capability of the musculature of the workers.

o power law of learning

15.1 Knowledge

A distinction is often made between different kinds of knowledge. Declarative knowledge are the facts; procedural knowledge are the skills (the ability to perform learned actions). Implicit memory is defined as

developer knowledge

implicit
learning

memory without conscious awareness—it might be considered a kind of knowledge.

15.1.1 Declarative knowledge

declarative knowl-
edge

This consists of knowledge about facts and events. For instance, the keywords used to denote the integer types are **char**, **short**, **int**, and **long**. This kind of knowledge is usually explicit (we know what we know), but there are situations where it can be implicit (we make use of knowledge that we are not aware of having^[262]). The coding guideline recommendations in this book have the form of declarative knowledge.

It is the connections and relationships between the individual facts, for instance the relative sizes of the integer types, that differentiate experts from novices (who might know the same facts). This kind of knowledge is rather like web pages on the Internet; the links between different pages corresponding to the connections between facts made by experts. Learning a subject is more about organizing information and creating connections between different items than it is about remembering information in a rote-like fashion.

This was demonstrated in a study by McKeithen, Reitman, Ruster, and Hirtle,^[287] who showed that developers with greater experience with a language organized their knowledge of language keywords in a more structured fashion. Education can provide the list of facts, it is experience that provides the connections between them.

The term *knowledge base* is sometimes used to describe a collection of information and links about a given topic. The C Standard document is a knowledge base. Your author has a C knowledge base in his head, as do you the reader. This book is another knowledge base dealing with C. The difference between this book and the C Standard document is that it contains significantly more explicit links connecting items, and it also contains information on how the language is commonly implemented and used.

15.1.2 Procedural knowledge

procedural knowl-
edge

This consists of knowledge about how to perform a task; it is often implicit.

Knowledge can start off by being purely declarative and, through extensive practice, becomes procedural; for instance, the process of learning to drive a car. An experiment by Sweller, Mawer, and Ward^[441] showed how subjects' behavior during mathematical problem solving changed as they became more proficient. This suggested that some aspects of what they were doing had been proceduralized.

Some of the aspects of writing source code that can become proceduralized are discussed elsewhere.

developer
flow
automa-
tization

15.2 Education

developer
education

What effect does education have on people who go on to become software developers?

Education should not be thought of as replacing the rules that people use for understanding the world but rather as introducing new rules that enter into competition with the old ones. People reliably distort the new rules in the direction of the old ones, or ignore them altogether except in the highly specific domains in which they were taught.

Page 206 of Hol-
land et al.^[172]

Education can be thought of as trying to do two things (of interest to us here)—teach students skills (procedural knowledge) and providing them with information, considered important in the relevant field, to memorize (declarative knowledge). To what extent does education in subjects not related to software development affect a developer's ability to write software?

Some subjects that are taught to students are claimed to teach general reasoning skills; for instance, philosophy and logic. There are also subjects that require students to use specific reasoning skills, for instance statistics requires students to think probabilistically. Does attending courses on these subjects actually have any measurable effect on students' capabilities, other than being able to answer questions in an exam. That is, having acquired some skill in using a particular system of reasoning, do students apply it outside of the domain in which they learnt it? Existing studies have supplied a *No* answer to this question.^[289,323] This *No* was even found to apply to specific skills; for instance, statistics (unless the problem explicitly involves statistical thinking within the applicable domain) and logic.^[68]

A study by Lehman, Lempert, and Nisbett^[252] measured changes in students' statistical, methodological, and conditional reasoning abilities (about everyday-life events) between their first and third years. They

found that both psychology and medical training produced large effects on statistical and methodological reasoning, while psychology, medical, and law training produced effects on the ability to perform conditional reasoning. Training in chemistry had no effect on the types of reasoning studied. An examination of the skills taught to students studying in these fields showed that they correlated with improvements in the specific types of reasoning abilities measured. The extent to which these reasoning skills transferred to situations that were not everyday-life events was not measured. Many studies have found that in general people do not transfer what they have learned from one domain to another.

o expertise
transfer to another
domain

It might be said that passing through the various stages of the education process is more like a filter than a learning exercise. Those that already have the abilities being the ones that succeed.^[471] A well-argued call to arms to improve students' general reasoning skills, through education, is provided by van Gelder.^[470]

Good education aims to provide students with an overview of a subject, listing the principles and major issues involved; there may be specific cases covered by way of examples. Software development does require knowledge of general principles, but most of the work involves a lot of specific details: specific to the application, the language used, and any existing source code, while developers may have been introduced to the C language as part of their education. The amount of exposure is unlikely to have been sufficient for the building of any significant knowledge base about the language.

15.2.1 Learned skills

Education provides students with *learned knowledge*, which relates to the title of this subsection *learned skills*. Learning a skill takes practice. Time spent by students during formal education practicing their programming skills is likely to total less than 60 hours. Six months into their first development job they could very well have more than 600 hours of experience. Although students are unlikely to complete their education with a lot of programming experience, they are likely to continue using the programming beliefs and practices they have acquired. It is not the intent of this book to decry the general lack of good software development training, but simply to point out that many developers have not had the opportunity to acquire good habits, making the use of coding guidelines even more essential.

o developer
expertise

Can students be taught in a way that improves their general reasoning skills? This question is not directly relevant to the subject of this book; but given the previous discussion, it is one that many developers will be asking. Based on the limited research carried out to date the answer seems to be yes. Learning requires intense, quality practice. This would be very expensive to provide using human teachers, and researchers are looking at automating some of the process. Several automated training aids have been produced to help improve students' reasoning ability and some seem to have a measurable affect.^[471]

15.2.2 Cultural skills

Cultural skills include the use of language and category formation. Nisbett and Norenzayan^[326] provide an overview of culture and cognition. A more practical guide to cultural differences and communicating with people from different cultures, from the perspective of US culture, is provided by Wise, Hannaman, Kozumplik, Franke, and Leaver.^[495]

o developer
language and
culture
o catego-
rization
cultural differ-
ences

15.3 Creating experts

To become an expert a person needs motivation, time, economic resources, an established body of knowledge to learn from, and teachers to guide.

One motivation is to be the best, as in chess and violin playing. This creates the need to practice as much as others at that level. Ericsson found^[113] that four hours per day was the maximum concentrated training that people could sustain without leading to exhaustion and burnout. If this is the level of commitment, over a 10-year period, that those at the top have undertaken, then anybody wishing to become their equal will have to be equally committed. The quantity of practice needed to equal expert performance in less competitive fields may be less. One should ask of an expert whether they attained that title because they are simply as good as the best, or because their performance is significantly better than non-experts.

In many domains people start young, between three and eight in some cases,^[113] their parents' interest being critical in providing equipment, transport to practice sessions, and the general environment in which to

study.

An established body of knowledge to learn from requires that the domain itself be in existence and relatively stable for a long period of time. The availability of teachers requires a domain that has existed long enough for them to have come up through the ranks; and one where there are sufficient people interested in it that it is possible to make as least as much from teaching as from performing the task.

The research found that domains in which the performance of experts was not significantly greater than non-experts lacked one or more of these characteristics.

15.3.1 Transfer of expertise to different domains

Research has shown that expertise within one domain does not confer any additional skills within another domain.^[11] This finding has been duplicated for experts in real-world domains, such as chess, and in laboratory-created situations. In one series of experiments, subjects who had practiced the learning of sequences of digits (after 50–100 hours of practice they could commit to memory, and recall later, sequences containing more than 20 digits) could not transfer their expertise to learning sequences of other items.^[66]

15.4 Expertise as mental set

Software development is a new field that is still evolving at a rapid rate. Most of the fields in which expert performance has been studied are much older, with accepted bodies of knowledge, established traditions, and methods of teaching.

Sometimes knowledge associated with software development does not change wholesale. There can be small changes within a given domain; for instance, the move from K&R C to ISO C.

In a series of experiments Wiley,^[492] showed that in some cases non-experts could outperform experts within their domain. She showed that an expert's domain knowledge can act as a mental set that limits the search for a solution; the expert becomes fixated within the domain. Also, in cases where a new task does not fit the pattern of highly proceduralized behaviors of an expert, a novice's performance may be higher.

15.5 Software development expertise

Given the observation that in some domains the acknowledged experts do not perform significantly better than non-experts, we need to ask if it is possible that any significant performance difference could exist in software development. Stewart and Lusk^[432] proposed a model of performance that involves seven components. The following discussion breaks down expertise in software development into five major areas.

1. *Knowledge (declarative) of application domain.* Although there are acknowledged experts in a wide variety of established application domains, there are also domains that are new and still evolving rapidly. The use to which application expertise, if it exists, can be put varies from high-level design to low-level algorithmic issues (i.e., knowing that certain cases are rare in practice when tuning a time-critical section of code).
2. *Knowledge (declarative) of algorithms and general coding techniques.* There exists a large body of well-established, easily accessible, published literature about algorithms. While some books dealing with general coding techniques have been published, they are usually limited to specific languages, application domains (e.g., embedded systems), and often particular language implementations. An important issue is the rigor with which some of the coding techniques have been verified; it often leaves a lot to be desired, including the level of expertise of the author.
3. *Knowledge (declarative) of programming language.* The C programming language is regarded as an established language. Whether 25 years is sufficient for a programming language to achieve the status of being established, as measured by other domains, is an open question. There is a definitive document, the ISO Standard, that specifies the language. However, the sales volume of this document has been extremely low, and most of the authors of books claiming to teach C do not appear to have read the standard. Given this background, we cannot expect any established community of expertise in the C language to be very large.

4. *Ability (procedural knowledge) to comprehend and write language statements and declarations that implement algorithms.* Procedural knowledge is acquired through practice. While university students may have had access to computers since the 1970s, access for younger people did not start to occur until the mid 1980s. It is possible for developers to have had 25 years of software development practice.
5. *Development environment.* The development environment in which people have to work is constantly changing. New versions of operating systems are being introduced every few years; new technologies are being created and old ones are made obsolete. The need to keep up with development is a drain on resources, both in intellectual effort and in time. An environment in which there is a rapid turnover in applicable knowledge and skills counts against the creation of expertise.

Although the information and equipment needed to achieve a high-level of expertise might be available, there are several components missing. The motivation to become the best software developer may exist in some individuals, but there is no recognized measure of what *best* means. Without the measuring and scoring of performances it is not possible for people to monitor their progress, or for their efforts to be rewarded. While there is a demand for teachers, it is possible for those with even a modicum of ability to make substantial amounts of money doing (not teaching) development work. The incentives for good teachers are very poor.

Given this situation we would not expect to find large performance differences in software developers through training. If training is insufficient to significantly differentiate developers the only other factor is individual ability. It is certainly your author's experience— individual ability is a significant factor in a developer's performance.

Until the rate of change in general software development slows down, and the demand for developers falls below the number of competent people available, it is likely that ability will continue to be the dominant factor (over training) in developer performance.

15.6 Software developer expertise

Having looked at expertise in general and the potential of the software development domain to have experts, we need to ask how expertise might be measured in people who develop software. Unfortunately, there are no reliable methods for measuring software development expertise currently available. However, based on the previously discussed issues, we can isolate the following technical competencies (social competencies^[320] are not covered here, although they are among the skills sought by employers,^[30] and software developers have their own opinions^[255,421]):

- Knowledge (declarative) of application domain.
- Knowledge (declarative) of algorithms and general coding techniques.
- Knowledge (declarative) of programming languages.
- Cognitive ability (procedural knowledge) to comprehend and write language statements and declarations that implement algorithms (a specialized form of general analytical and conceptual thinking).
- Knowledge (metacognitive) about knowledge (i.e., judging the quality and quantity of one's expertise).

Your author has firsthand experience of people with expertise individually within each of these components, while being non-experts in all of the others. People with application-domain expertise and little programming knowledge or skill are relatively common. Your author once implemented the semantics phase of a CHILL (Communications HIgh Level Language) compiler and acquired expert knowledge in the semantics of that language. One day he was shocked to find he could not write a CHILL program without reference to some existing source code (to *refresh* his memory of general program syntax); he had acquired an extensive knowledge based on the semantics of the language, but did not have the procedural knowledge needed to write a program (the compiler was written in another language).^{0.6}

^{0.6}As a compiler writer, your author is sometimes asked to help fix problems in programs written in languages he has never seen before (how can one be so expert and not know every language?). He now claims to be an expert at comprehending programs written in unknown languages for application domains he knows nothing about (he is helped by the fact that few languages have any truly unique constructs).

A developer's knowledge of an application domain can only be measured using the norms of that domain. One major problem associated with measuring overall developer expertise is caused by the fact that different developers are likely to be working within different domains. This makes it difficult to cross correlate measurements.

A study at Bell Labs^[95] showed that developers who had worked on previous releases of a project were much more productive than developers new to a project. They divided time spent by developers into discovery time (finding out information) and work time (doing useful work). New project members spent 60% to 80% of their time in discovery and 20% to 40% doing useful work. Developers experienced with the application spent 20% of their time in discovery and 80% doing useful work. The results showed a dramatic increase in efficiency (useful work divided by total effort) from having been involved in one project cycle and less dramatic an increase from having been involved in more than one release cycle. The study did not attempt to separate out the kinds of information being sought during discovery.

Another study at Bell Labs^[305] found that the probability of a fault being introduced into an application, during an update, correlated with the experience of the developer doing the work. More experienced developers seemed to have acquired some form of expertise in an application that meant they were less likely to introduce a fault into it.

A study of development and maintenance costs of programs written in C and Ada^[504] found no correlation between salary grade (or employee rating) and rate of bug fix/add feature rate.

Your author's experience is that developers' general knowledge of algorithms (in terms of knowing those published in well-known text-books) is poor. There is still a strongly held view, by developers, that it is permissible for them to invent their own ways of doing things. This issue is only of immediate concern to these coding guidelines as part of the widely held, developers', belief that they should be given a free hand to write source as they see fit.

There is a group of people who might be expected to be experts in a particular programming languages—those who have written a compiler for it (or to be exact those who implemented the semantics phase of the compiler, anybody working on others parts [e.g., code generation] does not need to acquire detailed knowledge of the language semantics). Your author knows a few people who are C language experts and have not written a compiler for that language. Based on your author's experience of implementing several compilers, the amount of study needed to be rated as an expert in one computer language is approximately 3 to 4 hours per day (not even compiler writers get to study the language for every hour of the working day; there are always other things that need to be attended to) for a year. During that period, every sentence in the language specification will be read and analyzed in detail several times, often in discussion with colleagues. Generally developer knowledge of the language they write in is limited to the subset they learned during initial training, perhaps with some additional constructs learned while reading other developers' source or talking to other members of a project. The behavior of the particular compiler they use also colors their view of those constructs.

Expertise in the act of comprehending and writing software is hard to separate from knowledge of the application domain. There is rarely any need to understand a program without reference to the application domain it was written for. When computers were centrally controlled, before the arrival of desktop computers, many organizations offered a programming support group. These support groups were places where customers of the central computer (usually employees of the company or staff at a university) could take programs they were experiencing problems with. The staff of such support groups were presented with a range of different programs for which they usually had little application-domain knowledge. This environment was ideal for developing program comprehension skills without the need for application knowledge (your author used to take pride in knowing as little as possible about the application while debugging the presented programs). Such support groups have now been reduced to helping customers solve problems with packaged software. Environments in which pure program-understanding skills can be learned now seem to have vanished.

What developers do is discussed elsewhere. An expert developer could be defined as a person who is able to perform these tasks better than the majority of their peers. Such a definition is open-ended (how is

better defined for these tasks?) and difficult to measure. In practice, it is productivity that is the sought-after attribute in developers.

0 productivity
developer

Some studies have looked at how developers differ (which need not be the same as measuring expertise), including their:

- ability to remember more about source code they have seen,
- personality differences,
- knowledge of the computer language used, and
- ability to estimate the effort needed to implement the specified functionality.^[206]

0 developers
organized knowl-
edge
0 developer
personality

A study by Jørgensen and Sjøberg^[207] looked at maintenance tasks (median effort 16-work hours). They found that developers' skill in predicting maintenance problems improved during their first two years on the job; thereafter there was no correlation between increased experience (average of 7.7 years' development experience, 3.4 years on maintenance of the application) and increased skill. They attributed this lack of improvement in skill to a lack of learning opportunities (in the sense of deliberate practice and feedback on the quality of their work).

Job advertisements often specify that a minimum number of years of experience is required. Number of years is known not to be a measure of expertise, but it provides some degree of comfort that a person has had to deal with many of the problems that might occur within a given domain.

15.6.1 Is software expertise worth acquiring?

Most developers are not professional programmers any more than they are professional typists. Reading and writing software is one aspect of their job. The various demands on their time is such that most spend a small portion of their time writing software. Developers need to balance the cost of spending time becoming more skillful programmers against the benefits of possessing that skill. Experience has shown that software can be written by relatively unskilled developers. One consequence of this is that few developers ever become experts in any computer language.

When estimating benefit over a relatively short period of time, time spent learning more about the application domain frequently has a greater return than honing programming skills.

15.7 Coding style

As an Englishman, your author can listen to somebody talking and tell if they are French, German, Australian, or one of many other nationalities (and sometimes what part of England they were brought up in). From what they say, I might make an educated guess about their educational level. From their use of words like *cool*, *groovy*, and so on, I might guess age and past influences (young or ageing hippie).

coding guidelines
coding style

Source code written by an experienced developer sometimes has a recognizable style. Your author can often tell if a developer's previous language was Fortran, Pascal, or Basic. But he cannot tell if their previous language was Lisp or APL (anymore than he can distinguish regional US accents, nor can many US citizens tell the difference among an English, Scottish, Irish, or Australian accent), because he has not had enough exposure to those languages.

source code
accent

Is coding style a form of expertise (a coherent framework that developers use to express their thoughts), or is it a ragbag of habits that developers happen to have? Programs have been written that can accurately determine the authorship of C source code (success rates of 73% have been reported^[236]). These experiments used, in part, source code written by people new to software development (i.e., students). Later work using neural networks^[237] was able to get the failure rate down to 2%. That it was possible to distinguish programs written by very inexperienced developers suggests that style might simply be a ragbag of habits (these developers not having had time to put together a coherent way of writing source).

The styles used by inexperienced developers can even be detected after an attempt has been made to hide the original authorship of the source. Plagiarism is a persistent problem in many universities' programming courses and several tools have been produced that automatically detect source code plagiarisms.^[365,477]

One way for a developer to show mastery of coding styles would be to have the ability to write source using a variety of different styles, perhaps even imitating the style of others. The existing author analysis tools are being used to verify that different, recognizable styles were being used.

It was once thought (and still is by some people) that there is a correct way to speak. Received Pronunciation (as spoken on the BBC many years ago) was once promoted as correct usage within the UK.

Similarly, many people believe that source code can be written in a good style or a bad style. A considerable amount of time has been, and will probably continue to be, spent discussing this issue. Your authors' position is the following:

- Identifiable source code styles exist.
- It is possible for people to learn new coding styles.
- It is very difficult to explain style to non-expert developers.
- Learning a new style is sufficiently time-consuming, and the benefits are likely to be sufficiently small, that a developer is best advised to invest effort elsewhere.

Students of English literature learn how to recognize writing styles. There are many more important issues that developers need to learn before they reach the stage where learning about stylistic issues becomes worthwhile.

The phrase coding guidelines and coding style are sometimes thought of, by developers of as being synonymous. This unfortunate situation has led to coding guidelines acquiring a poor reputation. While recognizing the coding style does exist, they are not the subject of these coding guidelines. The term *existing practice* refers to the kinds of constructs often found in existing programs. Existing practice is dealt with as an issue in its own right, independent of any concept of style.

coding
guidelines
introduction

16 Human characteristics

Humans are not ideal machines, an assertion that may sound obvious. However, while imperfections in physical characteristics are accepted, any suggestion that the mind does not operate according to the laws of mathematical logic is rarely treated in the same forgiving way. For instance, optical illusions are accepted as curious anomalies of the eye/brain system; there is no rush to conclude that human eyesight is faulty.

Optical illusions are often the result of preferential biases in the processing of visual inputs that, in most cases, are beneficial (in that they simplify the processing of ecologically common inputs). In Figure 0.19, which of the two squares indicated by the arrows is the brighter one? Readers can verify that the indicated squares have exactly the same grayscale level. Use a piece of paper containing two holes, that display only the two squares pointed to.

This effect is not caused by low-level processing, by the brain, of the input from the optic nerve; it is caused by high-level processing of the scene (recognizing the recurring pattern and that some squares are within a shadow). Anomalies caused by this high-level processing are not limited to grayscales. The brain is thought to have specific areas dedicated to the processing of faces. The, so-called, *Thatcher illusion* is an example of this special processing of faces. The two faces in Figure 0.20 look very different; turn the page upside down and they look almost identical.

Music is another input stimulus that depends on specific sensory input/brain affects occurring. There is no claim that humans cannot hear properly, or that they should listen to music derived purely from mathematical principles.

Studies have uncovered situations where the behavior of human cognitive processes does not correspond to some generally accepted norm, such as Bayesian inference. However, it cannot be assumed that cognitive limitations are an adaptation to handle the physical limitations of the brain. There is evidence to suggest that some of these so-called cognitive limitations provide near optimal solutions for some real-world problems.^[169]

The ability to read, write, and perform complex mathematical reasoning are very recent (compared to several million years of evolution) cognitive skill requirements. Furthermore, there is no evidence to suggest

human character-
istics

evolutionary
psychology

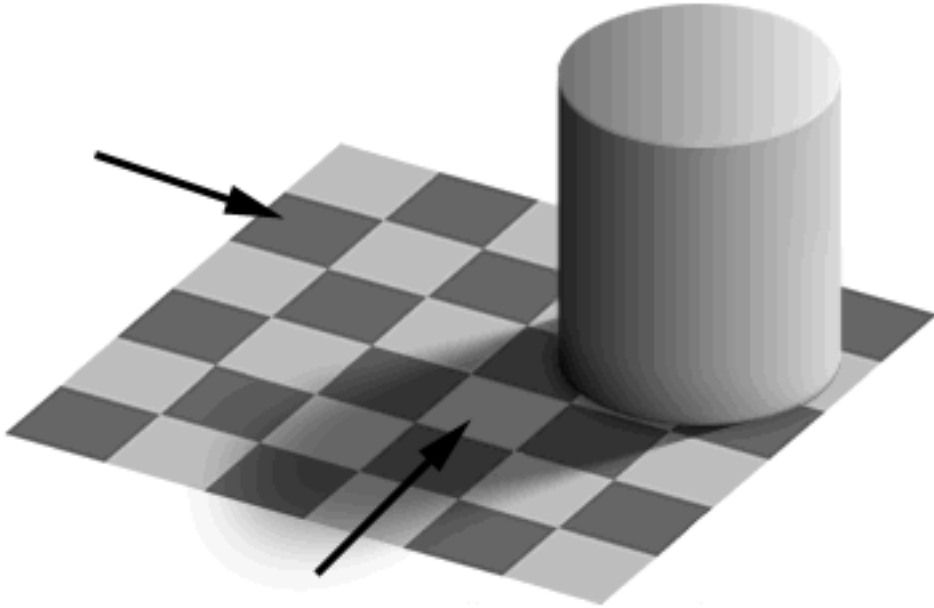


Figure 0.19: Checker shadow (by Edward Adelson). Which of the two squares indicated by the arrows is the brighter one (following inverted text gives answer)? Both squares reflect the same amount of light (this can be verified by collecting all of squares except the two indicated), but the human visual system assigns a relative brightness that is consistent with the checker pattern.



Figure 0.20: The Thatcher illusion. With permission from Thompson.^[453] The facial images look very similar when viewed in one orientation and very different when viewed in another (turn page upside down).

that possessing these skills improves the chances of a person passing on their genes to subsequent generations (in fact one recent trend suggests otherwise^[408]). So we should not expect human cognitive processes to be tuned for performing these activities.

Table 0.13: Cognitive anomalies. Adapted from McFadden.^[285]

Effect	Description
CONTEXT	
Anchoring	Judgments are influenced by quantitative cues contained in the statement of the decision task
Context	Prior choices and available options in the decision task influence perception and motivation
Framing	Selection between mathematically equivalent solutions to a problem depends on how their outcome is framed.
Prominence	The format in which a decision task is stated influences the weight given to different aspects
REFERENCE POINT	
Risk asymmetry	Subjects show risk-aversion for gains, risk-preference for losses, and weigh losses more heavily
Reference point	Choices are evaluated in terms of changes from an endowment or status quo point
Endowment	Possessed goods are valued more highly than those not possessed; once a function has been written
developers are loath to throw it away and start again	
AVAILABILITY	
Availability	Responses rely too heavily on readily retrievable information and too little on background information
Certainty	Sure outcomes are given more weight than uncertain outcomes
Experience	Personal history is favored relative to alternatives not experienced
Focal	Quantitative information is retrieved or reported categorically
Isolation	The elements of a multiple-part or multi-stage lottery are evaluated separately
Primacy and Recency	Initial and recently experienced events are the most easily recalled
Regression	Idiosyncratic causes are attached to past fluctuations, and regression to the mean is underestimated
Representativeness	High conditional probabilities induce overestimates of unconditional probabilities
Segregation	Lotteries are decomposed into a sure outcome and a gamble relative to this sure outcome
SUPERSTITION	
Credulity	Evidence that supports patterns and causal explanations for coincidences is accepted too readily
Disjunctive	Consumers fail to reason through or accept the logical consequences of actions
Superstition	Causal structures are attached to coincidences, and "quasi-magical" powers to opponents
Suspicion	Consumers mistrust offers and question the motives of opponents, particularly in unfamiliar situations
PROCESS	
Rule-Driven	Behavior is guided by principles, analogies, and exemplars rather than utilitarian calculus
Process	Evaluation of outcomes is sensitive to process and change
Temporal	Time discounting is temporally inconsistent, with short delays discounted too sharply relative to long delays
PROJECTION	
Misrepresentation	Subjects may misrepresent judgments for real or perceived strategic advantage
Projection	Judgments are altered to reinforce internally or project to others a self-image

Table 0.13 lists some of the cognitive anomalies (difference between human behavior and some idealized norm) applicable to writing software. There are other cognitive anomalies, some of which may also be applicable, and others that have limited applicability; for instance, writing software is a private, not a social activity. Cognitive anomalies relating to herd behavior and conformity to social norms are unlikely to be of

interest.

16.1 Physical characteristics

Before moving on to the main theme of this discussion, something needs to be said about physical characteristics.

The brain is the processor that the software of the mind executes on. Just as silicon-based processors have special units that software can make use of (e.g., floating point), the brain appears to have special areas that perform specific functions.^[348] This book treats the workings of the brain/mind combination as a black box. We are only interested in the outputs, not the inner workings (brain-imaging technology has not yet reached the stage where we can deduce functionality by watching the signals travelling along neurons).

Eyes are the primary information-gathering sensors for reading and writing software. A lot of research has been undertaken on how the eyes operate and interface with the brain.^[334] Use of other information-gathering sensors has been proposed, hearing being the most common (both spoken and musical^[480]). These are rarely used in practice, and they are not discussed further in this book.

Hands/fingers are the primary output-generation mechanism. A lot of research on the operation of limbs has been undertaken. The impact of typing on error rate is discussed elsewhere.

Developers are assumed to be physically mature (we do not deal with code written by children or adolescents) and not to have any physical (e.g., the impact of dyslexia on reading source code is not known; another unknown is the impact of deafness on a developer's ability to abbreviate identifiers based on their sound) or psychiatric problems.

Issues such as genetic differences (e.g., male vs. female^[359]) or physical changes in the brain caused by repeated use of some functional unit (e.g., changes in the hippocampi of taxi drivers^[276]) are not considered here.

16.2 Mental characteristics

This section provides an overview of those mental characteristics that might be considered important in reading and writing software. Memory, particularly short-term memory, is an essential ability. It might almost be covered under physical characteristics, but knowledge of its workings has not quite yet reached that level of understanding. An overview of the characteristics of memory is given in the following subsection. The consequences of these characteristics are discussed throughout the book.

The idealization of developers aspiring to be omnipotent logicians gets in the way of realistically approaching the subject of how best to make use of the abilities of the human mind. Completely rational, logical, and calculating thought may be considered to be the ideal tools for software development, but they are not what people have available in their heads. Builders of bridges do not bemoan the lack of unbreakable materials available to them, they have learned how to work within the limitations of the materials available. This same approach is taken in this book, work with what is available.

This overview is intended to provide background rationale for the selection of, some, coding guidelines. In some cases, this results in recommendations against the use of constructs that people are likely to have problems processing correctly. In other cases this results in recommendations to do things in a particular way. These recommendations could be based on, for instance, capacity limitations, biases/heuristics (depending on the point of view), or some other cognitive factors.

Some commentators recommend that ideal developer characteristics should be promoted (such ideals are often accompanied by a list of tips suggesting activities to perform to help achieve these characteristics, rather like pumping iron to build muscle). This book contains no exhortations to try harder, or tips on how to become better developers through mental exercises. In this book developers are taken as they are, not some idealized vision of how they should be.

Hopefully the reader will recognize some of the characteristics described here in themselves. The way forward is to learn to deal with these characteristics, not to try to change what could turn out to be intrinsic properties of the human brain/mind.

Software development is not the only profession for which the perceived attributes of practitioners do not

developer
physical char-
acteristics

typing mis-
takes

developer
mental char-
acteristics
memory
developer

correspond to reality. Darley and Batson^[91] performed a study in which they asked subjects (theological seminary students) to walk across campus to deliver a sermon. Some of the subjects were told that they were late and the audience was waiting, the remainder were not told this. Their journey took them past a *victim* moaning for help in a doorway. Only 10% of subjects who thought they were late stopped to help the victim; of the other subjects 63% stopped to help. These results do not match the generally perceived behavior pattern of theological seminary students.

Most organizations do not attempt to measure mental characteristics in developer job applicants; unlike many other jobs for which individual performance can be an important consideration. Whether this is because of an existing culture of not measuring, lack of reliable measuring procedures, or fear of frightening off prospective employees is not known.

16.2.1 Computational power of the brain

One commonly used method of measuring the performance of silicon-based processors is to quote the number of instructions (measured in millions) they can execute in a second. This is known to be an inaccurate measure, but it provides an estimate.

The brain might simply be a very large neural net, so there will be no instructions to count as such. Merkle^[291] used various approaches to estimate the number of synaptic operations per second; the followings figures are taken from his article:

- Multiplying the number of synapses (10^{15}) by their speed of operation (about 10 impulses/second) gives 10^{16} synapse operations per second.
- The retina of the eye performs an estimated 10^{10} analog add operations per second. The brain contains 10^2 to 10^4 times as many nerve cells as the retina, suggesting that it can perform 10^{12} to 10^{14} operations per second.
- A total brain power dissipation of 25 watts (an estimated 10 watts of useful work) and an estimated energy consumption of 5×10^{-15} joules for the switching of a nerve cell membrane provides an upper limit of 2×10^{15} operations per second.

A synapse switching on and off is rather like a transistor switching on and off. They both need to be connected to other switches to create a larger functional unit. It is not known how many synapses are used to create functional units in the brain, or even what those functional units might be. The distance between synapses is approximately 1 mm. Simply sending a signal from one part of the brain to another part requires many synaptic operations, for instance, to travel from the front to the rear of the brain requires at least 100 synaptic operations to propagate the signal. So the number of synaptic operations per high-level, functional operation is likely to be high. Silicon-based processors can contain millions of transistors. The potential number of transistor-switching operations per second might be greater than 10^{14} , but the number of instructions executed is significantly smaller.

Although there have been studies of the information-processing capacity of the brain (e.g., visual attention,^[478] storage rate into long-term memory,^[243] and correlations between biological factors and intelligence^[472]), we are a long way from being able to deduce the likely work rates of the components of the brain used during code comprehension. The issue of overloading the computational resources of the brain is discussed elsewhere.

There are several executable models of how various aspects of human cognitive processes operate. The ACT-R model^[13] has been applied to a wide range of problems, including learning, the visual interface, perception and action, cognitive arithmetic, and various deduction tasks.

Developers are familiar with the idea that a more powerful processor is likely to execute a program more quickly than a less powerful one. Experience shows that some minds are quicker at solving some problems than other minds and other problems (a correlation between what is known as *inspection time* and IQ has been found^[97]). For these coding guidelines, speed of mental processing is not a problem in itself. The problem of limited processing resources operating in a time-constrained environment, leading to errors being

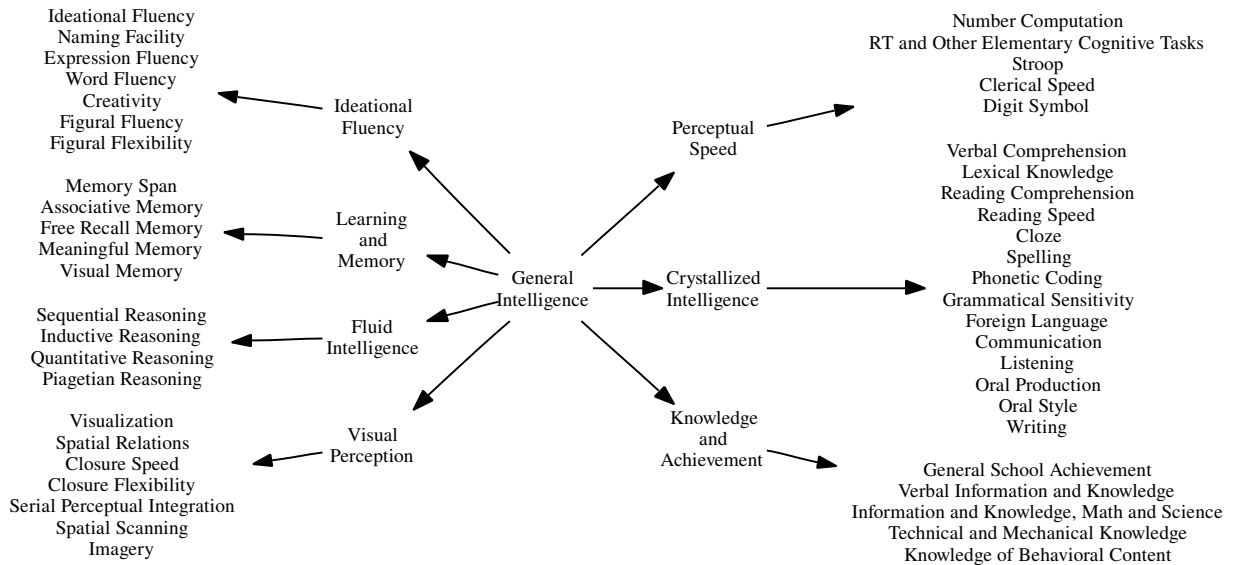


Figure 0.21: A list of and structure of ability constructs. Adapted from Ackerman.^[1]

made, could be handled if the errors were easily predicted. It is the fact that different developers have ranges of different abilities that cause the practical problems. Developer A can have trouble understanding the kinds of problems another developer, B, could have understanding the code he, A, has written. The problem is how does a person, who finds a particular task easy, relate to the problems a person, who finds that task difficult, will have?

The term *intelligence* is often associated with performance ability (to carry out some action in a given amount of time). There has been a great deal of debate about what intelligence is, and how it can be measured. Gardner^[138] argues for the existence of at least six kinds of intelligence—bodily kinesthetic, linguistic, mathematical, musical, personal, and spatial. Studies have shown that there can be dramatic differences between subjects rated high and low in these intelligences (linguistic^[148] and spatial^[273]). Ackerman and Heggstad^[1] review the evidence for overlapping traits between intelligence, personality, and interests (see Figure 0.21). An extensive series of tests carried out by Süß, Oberauer, Wittmann, Wilhelm, and Schulze^[437] found that intelligence was highly correlated to working memory capacity. The strongest relationship was found for reasoning ability.

The failure of so-called intelligence tests to predict students' job success on leaving college or university is argued with devastating effect by McClelland,^[282] who makes the point that the best testing is criterion sampling (for developers this would involve testing those attributes that distinguish *betterness* in developers). Until employers start to measure those employees who are involved in software development, and a theory explaining how these relate to the problem of developing software-based applications is available, there is little that can be said. At our current level of knowledge we can only say that developers having different abilities may exhibit different failure modes when solving problems.

16.2.2 Memory

Studies have found that human memory might be divided into at least two (partially connected) systems, commonly known as *short-term memory* (STM) and *long-term memory* (LTM). The extent to which STM and LTM really are different memory systems, and not simply two ends of a continuum of memory properties, continues to be researched and debated. Short-term memory tends to operate in terms of speech sounds and have a very limited capacity; while long-term memory tends to be semantic- or episodic-based and is often treated as having an infinite capacity (a lifetime of memories is estimated to be represented in 10^9 bits;^[243]

memory
developer

memory
episodic

this figure takes forgetting into account).

There are two kinds of query that are made against the contents of memory. During *recall* a person attempts to use information immediately available to them to access other information held in memory. During *recognition*, a person decides whether they have an existing memory for information that is being presented.

Much of the following discussion involves human memory performance with unchanging information. Developers often have to deal with changing information (e.g., the source code may be changing on a daily basis; the value of variables may be changing as developers run through the execution of code in their heads). Human memory performance has some characteristics that are specific to dealing with changing information.^[87,216] However, due to a lack of time and space, this aspect of developer memory performance is not covered in any detail in this book.

As its name implies, STM is an area of memory that stores information for short periods of time. For more than 100 years researchers have been investigating the properties of STM. Early researchers started by trying to measure its capacity. A paper by Miller^[296] entitled *The magical number seven, plus or minus two: Some limits on our capacity for processing information* introduced the now-famous 7 ± 2 rule. Things have moved on, during the 47 years since the publication of his paper^[204] (not that Miller ever proposed 7 ± 2 as the capacity of STM; he simply drew attention to the fact that this range of values fit the results of several experiments).

Readers might like to try measuring their STM capacity. Any Chinese-speaking readers can try this exercise twice, using the English and Chinese words for the digits.^[174] Use of Chinese should enable readers to apparently increase the capacity of STM (explanation follows). The digits in the outside margin can be used. Slowly and steadily read the digits in a row, out loud. At the end of each row, close your eyes and try to repeat the sequence of digits in the same order. If you make a mistake, go on to the next row. The point at which you cannot correctly remember the digits in any two rows of a given length indicates your capacity limit—the number of digits in the previous rows.

Measuring working memory capacity using sequences of digits relies on several assumptions. It assumes that working memory treats all items the same way (what if letters of the alphabet had been used instead), and it also assumes that individual concepts are the unit of storage. Studies have shown that both these assumptions are incorrect. What the preceding exercise measured was the amount of *sound* you could keep in working memory. The sound used to represent digits in Chinese is shorter than in English. The use of Chinese should enable readers to maintain information on more digits (average 9.9^[175]) using the same amount of sound storage. A reader using a language for which the sound of the digits is longer would be able to maintain information on fewer digits (e.g., average 5.8 in Welsh^[108]). The average for English is 6.6.

Studies have shown that performance on the *digit span* task is not a good predictor of performance on other short- or long-term memory for items. However, a study by Martin^[278] found that it did correlate with memory for the temporal occurrence of events.

In the 1970s Baddeley asked what purpose short-term memory served. He reasoned that its purpose was to act as a temporary area for activities such as mental arithmetic, reasoning, and problem solving. The model of *working memory* he proposed is shown in Figure 0.22. There are three components, each with its own independent temporary storage areas, each holding and using information in different ways.

What does the central executive do? It is assumed to be the system that handles attention, controlling the phonological loop, the visuo-spatial sketch pad, and the interface to long-term memory. The central executive needs to remember information while performing tasks such as text comprehension and problem solving. The potential role of this central executive is discussed elsewhere.

Visual information held in the visuo-spatial sketch pad decays very rapidly. Experiments have shown that people can recall four or five items immediately after they are presented with visual information, but that this recall rate drops very quickly after a few seconds. From the source code reading point of view, the visuo-spatial sketch pad is only operative for the source code currently being looked at.

While remembering digit sequences, readers may have noticed that the sounds used for them went around

Miller
7±2memory
digit span8704
2193
3172
57301
02943
73619
659420
402586
542173
6849173
7931684
3617458
27631508
81042963
07239861
578149306
293486701
721540683
5762083941
4093067215
9261835740Sequences of
single digits
containing 4
to 10 digits.attention 0
visuo-spatial
memory

phonological loop

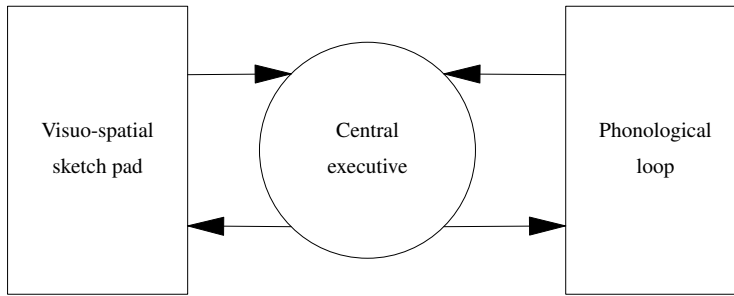


Figure 0.22: Model of working memory. Adapted from Baddeley.^[25]

in their heads. Research has uncovered a system known as the phonological (or articulatory) loop. This kind of memory can be thought of as being like a loop of tape. Sounds can be recorded onto this tape, overwriting the previous contents, as it goes around and around. An example of the functioning of this loop can be found, by trying to remember lists of words that vary by the length of time it takes to say them.

Table 0.14 contains lists of words; those at the top of the table contain a single syllable, those at the bottom multiple syllables. Readers should have no problems remembering a sequence of five single-syllable words, a sequence of five multi-syllable words should prove more difficult. As before, read each word slowly out loud.

Table 0.14: Words with either one or more than one syllable (and thus varying in the length of time taken to speak).

List 1	List 2	List 3	List 4	List 5
one	cat	card	harm	add
bank	lift	list	bank	mark
sit	able	inch	view	bar
kind	held	act	fact	few
look	mean	what	time	sum
ability	basically	encountered	laboratory	commitment
particular	yesterday	government	acceptable	minority
mathematical	department	financial	university	battery
categorize	satisfied	absolutely	meaningful	opportunity
inadequate	beautiful	together	carefully	accidental

It has been found that fast talkers have better short-term memory. The connection is the phonological loop. Short-term memory is not limited by the number of items that can be held. The limit is the length of sound this loop can store, about two seconds.^[26] Faster talkers can represent more information in that two seconds than those who do not talk as fast.

An analogy between *phonological loop* and a loop of tape in a tape recorder, suggests the possibility that it might only be possible to extract information as it goes past a *read-out point*. A study by Sternberg^[430] looked at how information in the phonological loop could be accessed. Subjects were asked to hold a sequences of digits, for instance 4185, in memory. They were then asked if a particular digit was in the sequence being held. The time taken to respond yes/no was measured. Subjects were given sequences of different length to hold in memory. The results showed that the larger the number of digits subjects had to hold in memory, the longer it took them to reply (see Figure 0.23). The other result was that the time to respond was not affected by whether the answer was yes or no. It might be expected that a yes answer would enable the search to be terminated. This suggests that all digits were always being compared.

A study by Cavanagh^[65] found that different kinds of information, held in memory, has different search response times (see Figure 0.24).

A good example of using the different components of working memory is mental arithmetic; for example, multiply 23 by 15 without looking at this page. The numbers to be multiplied can be held in the phonological

memory span

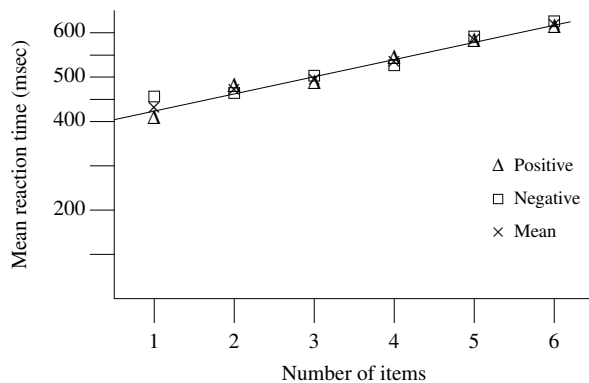


Figure 0.23: Judgment time (in milliseconds) as a function of the number of digits held in memory. Adapted from Sternberg.^[430]

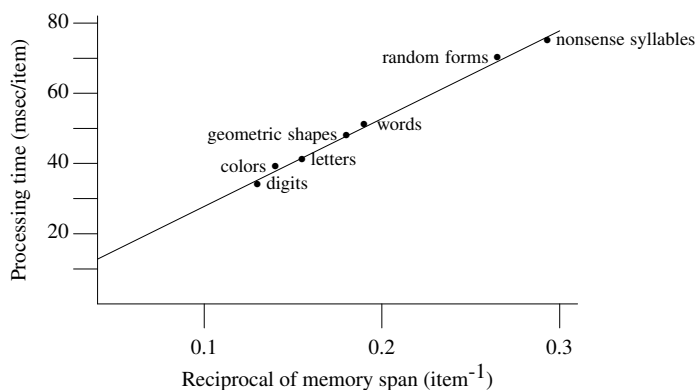


Figure 0.24: Judgment time (msec per item) as a function of the number of different items held in memory. Adapted from Cavanagh^[65]

loop, while information such as carries and which two digits to multiple next can be held within the central executive. Now perform another multiplication, but this time look at the two numbers being multiplied (see margin for values) while performing the multiplication.

While performing this calculation the visuo-spatial sketch pad can be used to hold some of the information, the values being multiplied. This frees up the phonological loop to hold temporary results, while the central executive holds positional information (used to decide which pairs of digits to look at). Carrying out a multiplication while being able to look at the numbers being multiplied seems to require less cognitive effort.

Recent research on working memory has begun to question whether it does have a capacity limit. Many studies have shown that people tend to organize items in memory in chunks of around four items. The role that attention plays in working memory, or rather the need for working memory in support of attention, has also come to the fore. It has been suggested that the focus of attention is capacity-limited, but that the other temporary storage areas are time-limited (without attention to rehearse them, they fade away). Cowan^[88] proposed the following:

1. The focus of attention is capacity-limited.
2. The limit in this focus averages about four chunks in normal adult humans.
3. No other mental faculties are capacity-limited, although some are limited by time and susceptibility to interference.

26
12

Two numbers
to multiply.

attention o

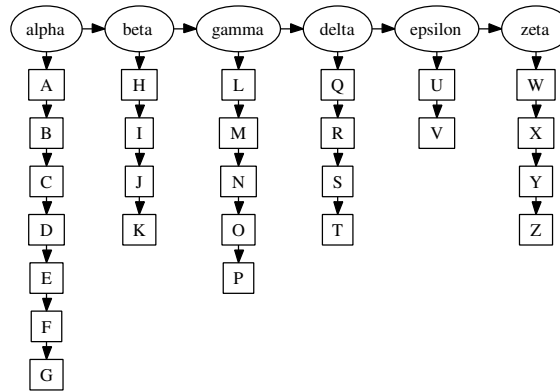


Figure 0.25: Semantic memory representation of alphabetic letters (the Greek names assigned to nodes by Klahr are used by the search algorithm and are not actually held in memory). Readers may recognize the structure of a nursery rhyme in the letter sequences. Derived from Klahr.^[222]

4. Any information that is deliberately recalled, whether from a recent stimulus or from long-term memory, is restricted to this limit in the focus of attention.

Other studies^[328] have used the results from multiple tasks to distinguish the roles (e.g., storage, processing, supervision, and coordination) of different components of working memory.

Chunking is a technique commonly used by people to help them remember information. A chunk is a small set of items (4 ± 1 is seen in many studies) having a common, strong, association with each other (and a much weaker one to items in other chunks). For instance, Wickelgren^[490] found that people's recall of telephone numbers is optimal if numbers are grouped into chunks of three digits. An example from random-letter sequences is *fbicbsibmirs*. The trigrams (*fb*, *cbs*, *ibm*, *irs*) within this sequence of 12 letters are well-known acronyms. A person who notices this association can use it to aid recall. Several theoretical analyses of memory organizations have shown that chunking of items improves search efficiency (^[101] optimal chunk size 3–4), (^[271] number items at which chunking becomes more efficient than a single list, 5–7).

An example of chunking of information is provided by a study performed by Klahr, Chase, and Lovelace^[222] who investigated how subjects stored letters of the alphabet in memory. Through a series of time-to-respond measurements, where subjects were asked to name the letter that appeared immediately before or after the presented probe letter, they proposed the alphabet-storage structure shown in Figure 0.25. They also proposed two search algorithms that described the process subjects used to answer the before/after question.

One of the characteristics of human memory is that it has knowledge of its own knowledge. People are good at judging whether they know a piece of information or not, even if they are unable to recall that information at a particular instant. Studies have found that so-called *feeling of knowing* is a good predictor of subsequent recall of information (see Koriat^[231] for a discussion and a model).

Several models of working memory are based on it only using a phonological representation of information. Any semantic effects in short-term memory come from information recalled from long-term memory. However, a few models of short-term memory do include a semantic representation of information (see Miyake and Shah^[303] for detailed descriptions of all the current models of working memory, and Baddeley for a comprehensive review^[23]).

A study by Hambrick and Engle^[158] asked subjects to remember information relating to baseball games. The subjects were either young, middle age, or old adult who knew little about baseball or were very knowledgeable about baseball. The largest factor (54.9%) in the variance of subject performance was expertise, with working memory capacity and age making up the difference.

Source code constructs differ in their likelihood of forming semantically meaningful chunks. For instance,

memory chunking

feeling of knowing

working memory information representation phonology

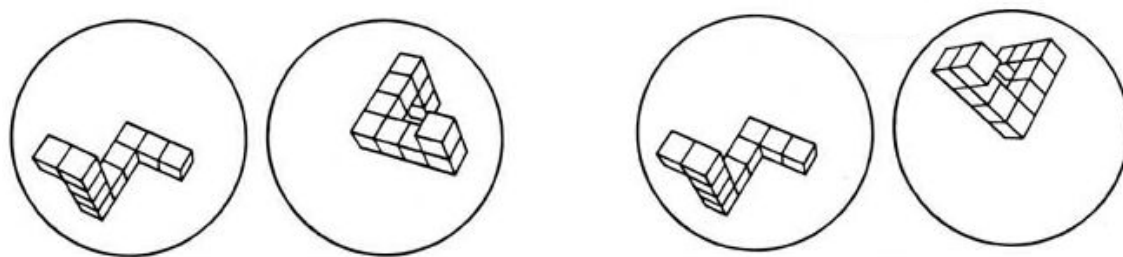


Figure 0.26: One of the two pairs are rotated copies of each other.

the ordering of a sequence of statements is often driven by the operations performed by those statements, while the ordering of parameters is often arbitrary.

Declarative memory is a long-term memory (information may be held until a person dies) that has a huge capacity (its bounds are not yet known) and holds information on facts and events (declarative knowledge is discussed elsewhere). Two components of declarative memory of interest to the discussion here are episodic and semantic memory. Episodic memory^[24] is a past-oriented memory system concerned with *remembering*, while semantic memory is a present-oriented memory system concerned with *knowing*.

Having worked on a program, a developer may remember particular sections of source code through their interaction with it (e.g., deducing how it interacted with other source code, or inserting traces to print out values of objects referenced in the code). After working on the same program for an extended period of time, a developer is likely to be able to recall information about it without being able to remember exactly when they learned that information.^[166]

16.2.2.1 Visual manipulation

How are visual images held in the brain? Are they stored directly in some way (like a bitmap), or are they held using an abstract representation (e.g., a list of objects tagged with their spatial positions). A study performed by Shepard^[401] suggested the former. He showed subjects pairs of figures and asked them if they were the same. Some pairs were different, while others were the same but had been rotated relative to each other. The results showed a linear relationship between the angle of rotation (needed to verify that two objects were the same) and the time taken to make a matching comparison. Readers might like to try their mind at rotating the pairs of images in Figure 0.26 to find out if they are the same.

Kosslyn^[233] performed a related experiment. Subjects were shown various pictures and asked questions about them. One picture was of a boat. Subjects were asked a question about the front of the boat and then asked a question about the rear of the boat. The response time, when the question shifted from the front to the rear of the boat, was longer than when the question shifted from one about portholes to one about the rear. It was as if subjects had to scan their image of the boat from one place to another to answer the questions.

A study by Presson and Montello^[366] asked two groups of subjects to memorize the locations of objects in a room. Both groups of subjects were then blindfolded and asked to point to various objects. The results showed their performance to be reasonably fast and accurate. Subjects in the first group were then asked to imagine rotating themselves 90°, then they were asked to point to various objects. The results showed their performance to be much slower and less accurate. Subjects in the second group were asked to actually rotate 90°; while still blindfolded, they were then asked to point to various objects. The results showed that the performance of these subjects was as good as before they rotated. These results suggest that mentally keeping track of the locations of objects, a task that many cognitive psychologists would suspect as being cognitive and divorced from the body, is in fact strongly affected by literal body movements (this result is more evidence for *the embodied mind* theory^[476] of the human mind).

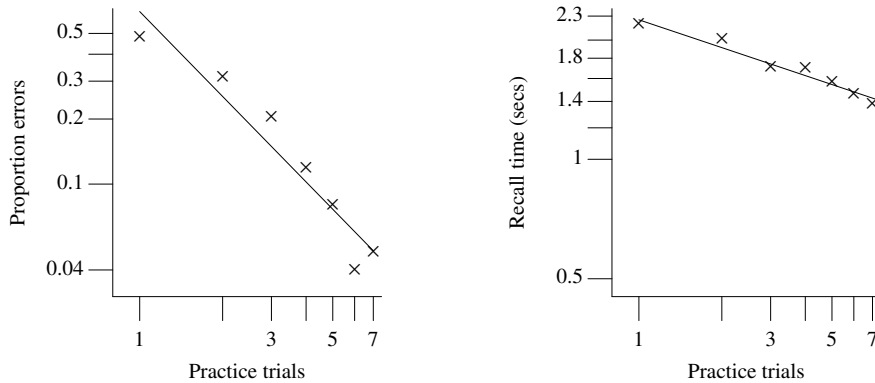


Figure 0.27: Proportion of errors (left) and time to recall (right) for recall of paired associate words (log scale). Based on Anderson.^[10]

16.2.2.2 Longer term memories

People can store large amounts of information for long periods of time in their long-term memory. Landauer^[243] attempts to estimate the total amount of learned information in LTM. Information written to LTM may not be held there for very long (storage), or it may be difficult to find (retrieval). This section discusses storage and retrieval of information in LTM.

One of the earliest memory research results was that practicing an item, after it had been learned, improves performance of recall at a later time (first published by Ebbinghaus in 1885, and reprinted several times since^[104]). The relationship between practice, P , and time, T , to recall has been found to follow a power law $T = aP^b$ (where a and b are constants). This relationship has become known as *the power law of learning*. A similar relationship has been found for error rates— more practice, fewer errors.

How is information stored in LTM? The brain contains neurons and synapses; information can only be represented as some kind of change in their state. The term *memory trace* is used to describe this changed state, representing the stored information. Accessing an information item in LTM is thought to increase the strength of its associated *memory trace* (which could mean that a stronger signal is returned by subsequent attempts at recall, or that the access path to that information is smoothed; nobody knows yet).

Practice is not the only way of improving recall. How an item has been studied, and its related associations, can affect how well it is recalled later. The meaning of information to the person doing the learning, so-called *depth of processing*, can affect their recall performance. Learning information that has a meaning is thought to create more access methods to its storage location(s) in LTM.

The *generation effect* refers to the process whereby people are involved in the generation of the information they need to remember. A study by Slamecka and Graf^[418] asked subjects to generate a synonym, or rhyme, synonym of a target word that began with a specified letter. For instance, generate a synonym for *sea* starting with the letter *o* (e.g., *ocean*). The subjects who had to generate the associated word showed a 15% improvement in recall, compared to subjects who had simply been asked to read the word pair (e.g., *sea–ocean*).

An example of the effect of additional, meaningful information was provided by a study by Bradshaw and Anderson.^[51] Subjects were given information on famous people to remember. For instance, one group of subjects was told:

Newton became emotionally unstable and insecure as a child

while other groups were given two additional facts to learn. These facts either elaborated on the first sentence or were unrelated to it:

memory
information
elaboration

Newton became emotionally unstable and insecure as a child
 Newton's father died when he was born
 Newton's mother remarried and left him with his grandfather

After a delay of one week, subjects were tested on their ability to recall the target sentence. The results showed that subjects percentage recall was higher when they had been given two additional sentences, that elaborated on the first one (the performance of subjects given related sentences being better than those given unrelated ones). There was no difference between subjects, when they were presented with the original sentence and asked if they recognized it.

The preceding studies involved using information that had a verbal basis. A study by Standing, Conezio, and Haber^[428] involved asking subjects to remember visual information. The subjects were shown 2,560 photographs for 10 seconds each (640 per day over a 4-day period). On each day, one hour after being shown the pictures, subjects were shown a random sample of 70 pairs of pictures (one of which was in the set of 640 seen earlier). They had to identify which of the pair they had seen before. Correct identification exceeded 90%. This and other studies have confirmed people's very good memory for pictures.

16.2.2.3 Serial order

The order in which items or events occur is often important when comprehending source code. For instance, the ordering of a function's parameters needs to be recalled when passing arguments, and the order of statements within the source code of a function specifies an order of events during program execution. Two effects are commonly seen in human memory recall performance:

1. The *primacy effect* refers to the better recall performance for items at the start of a list.
2. The *recency effect* refers to the better recall performance for items at the end of a list.

A number of models have been proposed to explain people's performance in the serial list recall task. Henson^[165] describes the *start–end model*.

16.2.2.4 Forgetting

While people are unhappy about the fact that they forget things, never forgetting anything may be worse. The Russian mnemonist Shereshevskii found that his ability to remember everything, cluttered up his mind.^[270] Having many similar, not recently used, pieces of information matching during a memory search would be counterproductive; forgetting appears to be a useful adaptation. For instance, a driver returning to a car wants to know where it was last parked, not the location of all previous places where it was parked. Anderson and Milson^[14] proposed that human memory is optimized for information retrieval based on the statistical properties of information use, in people's everyday lives; their work was based on a model developed by Burrell^[57] (who investigated the pattern of book borrowings in several libraries; which were also having items added to their stock). The rate at which the mind forgets seems to mirror the way that information tends to lose its utility in the real world over time.

It has only recently been reliably established^[387] that forgetting, like learning, follows a power law (the results of some studies could be fitted using exponential functions). The general relationship between the retention of information, R , and the time, T , since the last access has the form $R = aD^{-b}$ (where a and b are constants). It is known as the *power law of forgetting*. The constant a depends on the amount of initial learning. A study by Bahrick^[28] (see Figure 0.28) looked at subjects' retention of English–Spanish vocabulary (the drop-off after 25 years may be due to physiological deterioration^[11]).

The following are three theories of how forgetting occurs:

1. Memory traces simply fade away.
2. Memory traces are disrupted or obscured by newly formed memory traces created by new information being added to memory.

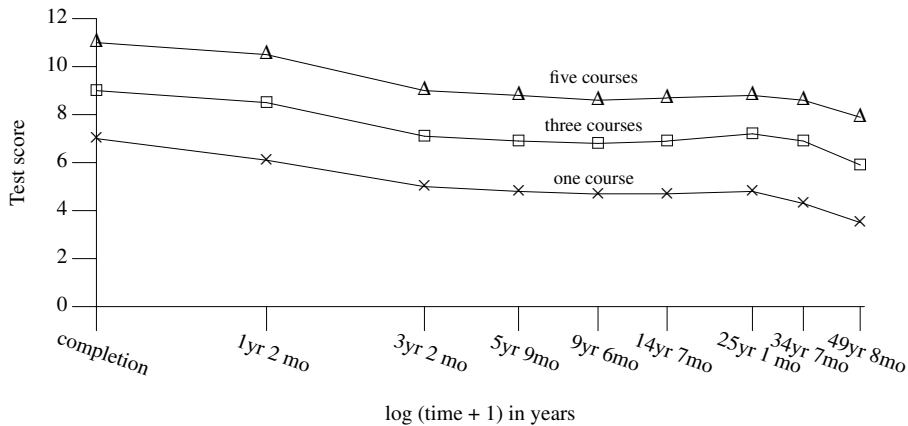


Figure 0.28: Effect of level of training on the retention of recognition of English-Spanish vocabulary. Adapted from Bahrick.^[28]

3. The retrieval cues used to access memory traces are lost.

The process of learning new information is not independent of already-learned information. There can be mutual inference between the two items of information. The interference of old information, caused by new information, is known as *retroactive interference*. It is not yet known whether the later information weakens the earlier information, or whether it is simply stronger and overshadows access to the earlier information. The opposite effect of retroactive interference is *proactive interference*. In this case, past memories interfere with more recent ones.

Table 0.15 and Table 0.16 (based on Anderson^[12]) are examples of the word-pair association tests used to investigate interference effects. Subjects are given a single pair of words to learn and are tested on that pair only (in both tables, Subject 3 is the control). The notation $A \Rightarrow B$ indicates that subjects have to learn to respond with B when given the cue A . An example of a word-pair is *sailor-tipsy*. The Worse/Better comparison is against the performance of the control subjects.

Table 0.15: Proactive inhibition. The third row indicates learning performance; the fifth row indicates recall performance, relative to that of the control. Based on Anderson.^[12]

Subject 1	Subject 2	Subject 3
Learn $A \Rightarrow B$	Learn $C \Rightarrow D$	Rest
Learn $A \Rightarrow D$	Learn $A \Rightarrow B$	Learn $A \Rightarrow D$
Worse	Better	
Test $A \Rightarrow D$	Test $A \Rightarrow D$	Test $A \Rightarrow D$
Worse	Worse	

Table 0.16: Retroactive inhibition. The fourth row indicates subject performance relative to that of the control. Based on Anderson.^[12]

Subject 1	Subject 2	Subject 3
Learn $A \Rightarrow B$	Learn $A \Rightarrow B$	Learn $A \Rightarrow B$
Learn $A \Rightarrow D$	Learn $C \Rightarrow D$	Rest
Test $A \Rightarrow B$	Test $A \Rightarrow B$	Test $A \Rightarrow B$
Much worse	Worse	

The general conclusion from the, many, study results is that interference occurs in both learning and recall

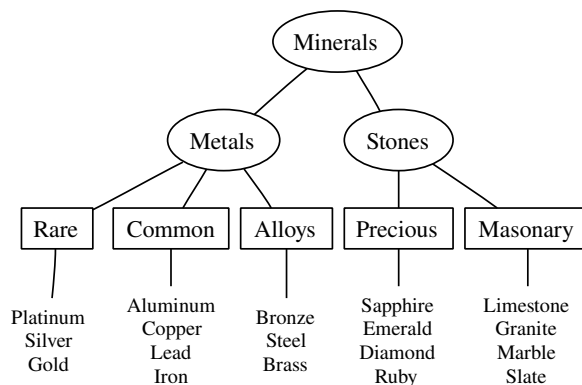


Figure 0.29: Words organized according to their properties—the *minerals* conceptual hierarchy. Adapted from Bower, Clark, Lesgold, and Winzenz.^[49]

when there are multiple associations for the same item. The improvement in performance of subjects in the second category, of proactive inhibition, is thought to occur because of a practice effect.

16.2.2.5 Organized knowledge

Information is not stored in people's LTM in an unorganized form (for a detailed discussion, see^[12]). This section provides a brief discussion of the issues. More detailed discussions are provided elsewhere in the specific cases that apply to reading and writing source code.

Whenever possible, the coding guidelines given in this book aim to take account of the abilities and limitations that developers have. An example of how it is possible to use an ability of the mind (organizing information in memory) to overcome a limitation (information held in LTM becoming inaccessible) is provided by the following demonstration.

Readers might like to try remembering words presented in an organized form and as a simple list. Read the words in Figure 0.29 out loud, slowly and steadily. Then try to recall as many as possible. Then repeat the process using the words given below. It is likely that a greater number of words will be recalled from the organized structure. The words in the second list could be placed into the same structure as the first list, instead they appear in a random order.

pine elm pansy garden wild banyan plants
 delphinium conifers dandelion redwood palm ash
 violet daisy tropical chestnut flowers spruce lupin
 buttercup trees deciduous mango willow rose

Familiarity with information being learned and recalled can also make a difference. Several studies have shown that experts perform better than non-experts in remembering information within their domain of expertise. For instance, McKeithen, Reitman, Ruster, and Hirtle^[287] measured developers' ability to memorize program source code. Subjects were presented with two listings; one consisted of a sequence of lines that made up a well-formed program, the other contained the same lines but the order in which they appeared on the listing had been randomized. Experienced developers (more than 2,000 hr of general programming and more than 400 hr experience with the language being used in the experiment) did a much better job at recalling source lines from the listing that represented a well-formed program and inexperienced developers. Both groups did equally well in recalling lines from the randomized listing. The experiments also looked at how developers remembered lists of language keywords they were given. How the information was organized was much more consistent across experienced developers than across inexperienced developers (experienced developers also had a slightly deeper depth of information chunking, 2.26 vs. 1.88).

16.2.2.6 Memory accuracy

Until recently experimental studies of memory have been dominated by a quantity-oriented approach. Memory was seen as a storehouse of information and is evaluated in terms of how many items could be successfully retrieved. The issue of accuracy of response was often ignored. This has started to change and there has been a growing trend for studies to investigate accuracy.^[232] Coding guidelines are much more interested in factors that affect memory accuracy than those, for instance, affecting rate of recall. Unfortunately, some of the memory studies described in this book do not include information on error rates.

16.2.2.7 Errors caused by memory overflow

Various studies have verified that limits on working memory can lead to an increase in a certain kind of error when performing a complex task. Byrne and Bovair^[58] looked at postcompletion errors (an example of this error is leaving the original in the photocopy machine after making copies, or the ATM card in the machine after withdrawing money) in complex tasks. A task is usually comprised of several goals that need to be achieved. It is believed that people maintain these goals using a stack mechanism in working memory. Byrne and Bovair were able to increase the probability of subjects making postcompletion errors in a task assigned to them. They also built a performance model that predicted postcompletion errors that were consistent with those seen in the experiments.

developer errors
memory overflow

The possible impact of working memory capacity-limits in other tasks, related to reading and writing source code, is discussed elsewhere. However, the complexity of carrying out studies involving working memory should not be underestimated. There can be unexpected interactions from many sources. A study by Lemaire, Abdi, and Fayol^[254] highlighted the complexity of trying to understand the affects of working memory capacity limitations. The existing models of the performance of simple arithmetic operations, involve an interrelated network in long-term memory (built during the learning of arithmetic facts, such as the multiplication table, and reinforced by constant practice). Lemaire et al. wanted to show that simple arithmetic also requires working memory resources.

conditional
statement

To show that working memory resources were required, they attempted to overload those resources. Subjects were required to perform another task at the same time as answering a question involving simple arithmetic (e.g., $4 + 8 = 12$, true or false?). The difficulty of the second task varied between experiments. One required subjects to continuously say the word *the*, another had them continuously say the letters *abcdef*, while the most difficult task required subjects to randomly generate letters from the set *abcdef* (this was expected to overload the central executive system in working memory).

The interesting part of the results (apart from confirming the authors' hypothesis that working memory was involved in performing simple arithmetic) was how the performances varied depending on whether the answer to the simple arithmetic question was true or false. The results showed that performance for problems that were true was reduced when both the phonological loop and the central executive were overloaded, while performance on problems that were false was reduced when the central executive was overloaded.

phonological
loop

A conditional expression requires that attention be paid to it if a developer wants to know under what set of circumstances it is true or false. What working memory resources are needed to answer this question; does keeping the names of identifiers in the phonological loop, or filling the visuo-spatial sketch pad (by looking at the code containing the expression) increase the resources required; does the semantics associated with identifiers or conditions affect performance? Your author does not know the answers to any of these questions but suspects that these issues, and others, are part of the effort cost that needs to be paid in extracting facts from source code.

16.2.2.8 Memory and code comprehension

As the results from the studies just described show, human memory is far from perfect. What can coding guidelines do to try to minimize potential problems caused by these limitations? Some authors of other coding guideline documents have obviously heard of Miller's^[296] 7 ± 2 paper (although few seem to have read it), often selecting five as the maximum bound on the use of some constructs.^[204] However, the effects of working memory capacity-limits cannot be solved by such simple rules. The following are some of the many issues that need to be considered:

- What code is likely to be written as a consequence of a guideline recommendation that specifies some limit on the use of a construct? Would following the guideline lead to code that was more difficult to comprehend?
- Human memory organizes information into related chunks (which can then be treated as a single item) multiple chunks may in turn be grouped together, forming a structured information hierarchy. The visibility of this structure in the visible source may be beneficial.
- There are different limits for different kinds of information.
- All of the constructs in the source can potentially require working memory resources. For instance, identifiers containing a greater number of syllables consume more resources in the phonological loop.

identifier
cognitive re-
source usage

There has been some research on the interaction between human memory and software development. For instance, Altmann^[7] built a computational process model based on SOAR, and fitted it to 10.5 minutes of programmer activity (debugging within an emacs window). The simulation was used to study the memories, called near-term memory by Altmann, built up while trying to solve a problem. However, the majority of studies discussed in this book are not directly related to reading and writing source code (your author has not been able to locate many). They can, at best, be used to provide indicators. The specific applications of these results occur throughout this book. They include reducing interference between information chunks and reducing the complexity of reasoning tasks.

identifier
syntax
selection
statement
syntax

16.2.2.9 Memory and aging

A study by Swanson^[440] investigated how various measures of working memory varied with the age of the subject. The results from diverse working memory tasks were reasonably intercorrelated. The following are the general conclusions:

- Age-related differences are better predicted by performance on tasks that place high demands on accessing information or maintaining old information in working memory than on measures of processing efficiency.
- Age-related changes in working memory appear to be caused by changes in a general capacity system.
- Age-related performance for both verbal and visuo-spatial working memory tasks showed similar patterns of continuous growth that peak at approximately age 45.

memory
ageing

visuo-spatial
memory

16.2.3 Attention

Attention is a limited resource provided by the human mind. It has been proposed that the age we live in is not the information age, but the attention age.^[93] Viewed in resource terms there is often significantly more information available to a person than attention resources (needed to process it). This is certainly true of the source code of any moderately large application.

Much of the psychology research on attention has investigated how inputs from our various senses handled. It is known that they operate in parallel and at some point there is a serial bottleneck, beyond which point it is not possible to continue processing input stimuli in parallel. The point at which this bottleneck occurs is a continuing subject of debate. There are early selection theories, late selection theories, and theories that combine the two.^[337] In this book, we are only interested in the input from one sense, the eyes. Furthermore, the scene viewed by the eyes is assumed to be under the control of the viewer. There are no objects that spontaneously appear or disappear; the only change of visual input occurs when the viewer turns a page or scrolls the source code listing on a display.

Read the bold print in the following paragraph:

Somewhere **Among** hidden **the** in **most** the **spectacular** Rocky Mountains **cognitive** near **abilities** Central City **is** Colorado **the** an **ability** old **to** miner **select** hid **one** a **message** box **from** of **another**. gold. **We** Although **do** several **this** hundred **by** people **focusing** have **our** looked **attention** for **on** it, **certain** they **cues** have **such** not **as** found **type** it **style**.

attention

What do you remember from the regular, non-bold, text? What does this tell you about selective attention?

People can also direct attention to their internal thought processes and memories. Internal thought processes are the main subject of this section. The issue of automatization (the ability to perform operations automatically after a period of training) is also covered; visual attention is discussed elsewhere.

o automatization

Ideas and theories of attention and conscious thought are often intertwined. While of deep significance, these issues are outside the scope of this book. The discussion in this section treats attention as a resource available to a developer when reading and writing source code. We are interested in knowing the characteristics of this resource, with a view to making the best use of the what is available. Studies involving attention have looked at capacity limits, the cost of changes of attention, and why some thought-conscious processes require more effort than others.

The following are two attention resource theories:

- *The single-capacity theory.* This proposes that performance depends on the availability of resources; more information processing requires more resources. When people perform more than one task at the same time, the available resources per task is reduced and performance decreases.
- *The multiple-resource theory.* This proposes that there are several different resources. Different tasks can require different resources. When people perform more than one task at the same time, the effect on the response for each task will depend on the extent to which they need to make use of the same resource at the same time.

Many of the multiple-resource theory studies use different sensory input tasks; for instance, subjects are required to attend to a visual and an audio channel at the same time. Reading source code uses a single sensory input, the eyes. However, the input is sufficiently complex that it often requires a great deal of thought. The extent to which code reading *thought* tasks are sufficiently different that they will use different cognitive resources is unknown. Unless stated otherwise, subsequent discussion of attention will assume that the tasks being performed, in a particular context, call on the same resources.

As discussed previously, the attention, or rather the focus of attention is believed to be capacity-limited. Studies suggest that this limit is around four chunks.^[88] Studies^[479] have also found that attention performance has an age-related component.

o memory developer

Power law of learning

Studies have found that nearly every task that exhibits a practice effect follows the *power law of learning*; which has the form:

power law of learning

$$RT = a + bN^{-c} \quad (0.21)$$

where RT is the response time; N is the number of times the task has been performed; and a , b , and c are constants. There were good theoretical reasons for expecting the equation to have an exponential form (i.e., $a + be^{-cN}$); many of the experimental results could be fitted to such an equation. However, if chunking is assumed to play a part in learning, a power law is a natural consequence (see Newell^[321] for a discussion).

16.2.4 Automatization

Source code contains several frequently seen patterns of usage. Experienced developers gain a lot of experience writing (or rather typing in) these constructs. As experience is gained, developers learn to type in these constructs without giving much thought to what they are doing. This process is rather like learning to write at school; children have to concentrate on learning to form letters and the combination of letters that form a word. After sufficient practice, many words only need to be briefly thought before they appear on the page without conscious effort.

o automatization culture of C

The *instance theory* of automatization^[266] specifies that novices begin by using an algorithm to perform a task. As they gain experience they learn specific solutions to specific problems. These solutions are retrieved

from memory when required. Given sufficient experience, the solution to all task-related problems can be obtained from memory and the algorithmic approach, to that task, is abandoned. The underlying assumptions of the theory are that encoding of problem and solution in memory is an unavoidable consequence of attention. Attending to a stimulus is sufficient to cause it to be committed to memory. The theory also assumes that retrieval of the solution from memory is an unavoidable consequence of attending to the task (this retrieval may not be successful, but it occurs anyway). Finally, each time the task is encountered (the instances) it causes encoding, storing, and retrieval, making it a learning-based theory.

Automatization (or automaticity) is an issue for coding guidelines in that many developers will have learned to use constructs whose use is recommended against. Developers' objections to having to stop using constructs that they know so well, and having to potentially invest in learning new techniques, is something that management has to deal with.

16.2.5 Cognitive switch

Some cognitive processes are controlled by a kind of *executive* mechanism. The nature of this executive is poorly understood and its characteristics are only just starting to be investigated.^[220] The process of comprehending source code can require switching between different tasks. Studies^[309] have found that subjects responses are slower and more error prone immediately after switching tasks. The following discussion highlights the broader research results.

A study by Rogers and Monsell^[383] used the two tasks of classifying a letter as a consonant or vowel, and classifying a digit as odd or even. The subjects were split into three groups. One group was given the latter classification task, the second group the digit classification task, and the third group had to alternate (various combinations were used) between letter and digit classification. The results showed that having to alternate tasks slowed the response times by 200 to 250 ms and the error rates went up from 2% to 3% to 6.5% to 7.5%. A study by Altmann^[8] found that when the new task shared many features in common with the previous task (e.g., switching from classifying numbers as odd or even, to classifying them as less than or greater than five) the memories for the related tasks interfered, causing a reduction in subject reaction time and an increase in error rate.

The studies to date have suggested the following conclusions:^[119]

- When it occurs the alternation cost is of the order of a few hundred milliseconds, and greater for more complex tasks.^[388]
- When the two tasks use disjoint stimulus sets, the alternation cost is reduced to tens of milliseconds, or even zero. For instance, the tasks used by Spector and Biederman^[423] were to subtract three from Arabic numbers and name antonyms of written words.
- Adding a cue to each item that allows subjects to deduce which task to perform reduces the alternation cost. In the Spector and Biederman study, they suffixed numbers with “+3” or “-3” in a task that required them to add or subtract three from the number.
- An alternation cost can be found in tasks having disjoint stimulus sets when those stimulus sets occurred in another pair of tasks that had recently been performed in alternation.

These conclusions raise several questions in a source code reading context. To what extent do different tasks involve different stimulus sets and how prominent must a cue be (i.e., is the 0x on the front of a hexadecimal number sufficient to signal a change of number base)? These issues are discussed elsewhere under the C language constructs that might involve cognitive task switches.

Probably the most extreme form of cognitive switch is an external interruption. In some cases, it may be necessary for developers to perform some external action (e.g., locating a source file containing a needed definition) while reading source code. Latorella^[247] discusses the impact of interruptions on the performance of flight deck personnel (in domains where poor performance in handling interruptions can have fatal consequences), and McFarlane^[286] provides a human-computer interruption taxonomy.

cognitive switch

antonym

character
constant
value
bitwise op-
erators

16.2.6 Cognitive effort

Why do some mental processes seem to require more mental effort than others? Why is effort an issue in mental operations? The following discussion is based on Chapter 8 of Pashler.^[337]

cognitive effort

One argument is that mental effort requires energy, and the body's reaction to concentrated thinking is to try to conserve energy by creating a sense of effort. Studies of blood flow show that the brain accounts for 20% of heart output, and between 20% to 25% of oxygen and glucose requirements. But, does concentrated thinking require a greater amount of metabolic energy than sitting passively? The answer from PET scans of the brain appears to be no. In fact the energy consumption of the visual areas of the brain while watching television are higher than the consumption levels of those parts of the brain associated with *difficult thinking*.

Another argument is that the repeated use of neural systems produces a temporary reduction in their efficiency. A need to keep these systems in a state of readiness (fight or flight) could cause the sensation of mental effort. The results of some studies are not consistent with this repeated use argument.

The final argument, put forward by Pashler, is that *difficult thinking* puts the cognitive system into a state where it is close to failing. It is the internal recognition of a variety of signals of impending cognitive failure that could cause the feeling of mental effort.

At the time of this writing there is no generally accepted theory of the root cause of cognitive effort. It is a recognized effect and developers' reluctance to experience it is a factor in the specification some of the guideline recommendations.

What are the components of the brain that are most likely to be resource limited when performing a source code comprehension task? Source code comprehension involves many of the learning and problem solving tasks that students encounter in the class room. Studies have found a significant correlation between the working memory requirements of a problem and students' ability to solve it^[427] and teenagers academic performance in mathematics and science subjects (but not English).^[139]

Most existing research has attempted to find a correlation between a subjects learning and problem solving performance and the capacity of their working memory.^[79] Some experiments have measured subjects recall performance, after performing various tasks. Others have measured subjects ability to make structure the information they are given into a form that enables them to answer questions about it^[157] (e.g., who met who in "The boy the girl the man saw met slept.").

Cognitive load might be defined as the total amount of mental activity imposed on working memory at any instant of time. The cognitive effort needed to solve a problem being the sum of all the cognitive loads experienced by the person seeking the solution.

cognitive load

$$Cognitive\ effort = \sum_{i=1}^t Cognitive\ load_i \quad (0.22)$$

Possible techniques for reducing the probability that a developers working memory capacity will be exceeded during code comprehension include:

- organizing information into chunks that developers are likely to recognize and have stored in their long-term memory,
- minimizing the amount of information that developers need to simultaneously keep in working memory during code comprehension (i.e., just in time information presentation),
- minimizing the number of relationships between the components of a problem that need to be considered (i.e., break it up into smaller chunks that can be processed independently of each other). Algorithms based on database theory and neural networks^[157] have been proposed as a method of measuring the *relational complexity* of a problem.

memory
chunking

identifier
cognitive resource
usage

16.2.7 Human error

developer errors

The discussion in this section has been strongly influenced by *Human Error* by Reason.^[375] Models of errors made by people have been broken down, by researchers, into different categories.

Table 0.17: Main failure modes for skill-based performance. Adapted from Reason.^[375]

Inattention	Over Attention
Double-capture slips	Omissions
Omissions following interruptions	Repetitions
Reduced intentionality	Reversals
Perceptual confusions	
Interference errors	

- *Skill-based* errors (see Table 0.17) result from some failure in the execution and/or the storage stage of an action sequence, regardless of whether the plan which guided when was adequate to achieve its objective. Those errors that occur during execution of an action are called *slips* and those that occur because of an error in memory are called *lapses*.
- *Mistakes* can be defined as deficiencies or failures in the judgmental and/or inferential processes involved in the selection of an objective or in the specification of the means to achieve it, irrespective of whether the actions directed by this decision-scheme run according to plan. *Mistakes* are further categorized into one of two kinds— *knowledge-based* mistakes (see Table 0.18) mistakes and *rule based* mistakes (see Table 0.19).

Table 0.18: Main failure modes for knowledge-based performance. Adapted from Reason.^[375]

Knowledge-based Failure Modes
Selectivity
Workspace limitations
Out of, sight out of mind
Confirmation bias
Overconfidence
Biased reviewing
Illusory correlation
Halo effects
Problems with causality
Problems with complexity
Problems with delayed feed-back
Insufficient consideration of processes in time
Difficulties with exponential developments
Thinking in causal series not causal nets (unaware of side-effects of action)
Thematic vagabonding (flitting from issue to issue)
Encysting (lingering in small detail over topics)

This categorization can be of use in selecting guideline recommendations. It provides a framework for matching the activities of developers against existing research data on error rates. For instance, developers would make skill-based errors while typing into an editor or using cut-and-paste to move code around.

Table 0.19: Main failure modes for rule-based performance. Adapted from Reason.^[375]

Misapplication of Good Rules	Application of Bad Rules
First exceptions	Encoding deficiencies
Countersigns and nosigns	Action deficiencies
Information overload	Wrong rules
Rule strength	Inelegant rules
General rules	Inadvisable rules
Redundancy	
Rigidity	

16.2.7.1 Skill-based mistakes

The consequences of possible skill-based mistakes may result in a coding guideline being created. However, by their very nature these kinds of mistakes cannot be directly recommended against. For instance, mistypings of identifier spellings leads to a guideline recommendation that identifier spellings differ in more than one significant character. A guideline recommending that identifier spellings not be mistyped being pointless.

?? identifier typed form

Information on instances of this kind of mistake can only come from experience. They can also depend on development environments. For instance, cut-and-paste mistakes may vary between use of line-based and GUI-based editors.

16.2.7.2 Rule-based mistakes

Use of rules to perform a task (a rule-based performance) does not imply that if a developer has sufficient expertise within the given area that they no longer need to expend effort thinking about it (a knowledge-based performance), only that a rule has been retrieved, from the memory, and a decision made to use it (rending a knowledge-based performance).

rule-base mistakes

The starting point for the creation of guideline recommendations intended to reduce the number of rule-based mistakes, made by developers is an extensive catalog of such mistakes. Your author knows of no such catalog. An indication of the effort needed to build such a catalog is provided by a study of subtraction mistakes, done by VanLehn.^[475] He studied the mistakes made by children in subtracting one number from another, and built a computer model that predicted many of the mistakes seen. The surprising fact, in the results, was the large number of diagnosed mistakes (134 distinct diagnoses, with 35 occurring more than once). That somebody can write a 250-page book on subtraction mistakes, and the model of procedural errors built to explain them, is an indication that the task is not trivial.

Holland, Holyoak, Nisbett, and Thagard^[172] discuss the use of rules in solving problems by induction and the mistakes that can occur through different rule based performances.

16.2.7.3 Knowledge-based mistakes

Mistakes that occur when people are forced to use a knowledge-based performance have two basic sources: bounded rationality and an incomplete or inaccurate mental model of the problem space.

o bounded rationality

A commonly used analogy of knowledge-based performances is that of a beam of light (working memory) that can be directed at a large canvas (the mental map of the problem). The direction of the beam is partially under the explicit control of its operator (the human conscious). There are unconscious influences pulling the beam toward certain parts of the canvas and avoiding other parts (which may, or may not, have any bearing on the solution). The contents of the canvas may be incomplete or inaccurate.

People adopt a variety of strategies, or heuristics, to overcome limitations in the cognitive resources available to them to perform a task. These heuristics appear to work well in the situations encountered in everyday human life, especially so since they are widely used by large numbers of people who can share in a common way of thinking.

Reading and writing source code is unlike everyday human experiences. Furthermore, the reasoning methods used by the non-carbon-based processor that executes software are wholly based on mathematical logic, which is only one of the many possible reasoning methods used by people (and rarely the preferred one at that).

There are several techniques for reducing the likelihood of making knowledge-based mistakes. For instance, reducing the size of the canvas that needs to be scanned and acknowledging the effects of heuristics.

16.2.7.4 Detecting errors

The modes of control for both skill-based and rule-based performances are feed-forward control, while the mode for knowledge-based performances is feed-back control. Thus, the detection of any skill-based or rule-based mistakes tends to occur as soon as they are made, while knowledge-based mistakes tend to be detected long after they have been made.

There have been studies looking at how people diagnose problems caused by knowledge-based mistakes.^[153] However, these coding guidelines are intended to provide advice on how to reduce the number of mistakes, not how to detect them once they have been made. Enforcement of coding guidelines to ensure that violations are detected is a very important issue.

16.2.7.5 Error rates

There have been several studies of the quantity of errors made by people performing various tasks. It is relatively easy to obtain this information for tasks that involve the creation of something visible (e.g., written material, of a file on a computer). Obtaining reliable error rates for information that is read and stored (or not) in people's memory is much harder to obtain. The following error rates may be applicable to writing source code:

- Touch typists, who are performing purely data entry:^[280] with no error correction 4% (per keystroke), typing nonsense words (per word) 7.5%.
- Typists using a line-oriented word processor:^[394] 3.40% of (word) errors were detected and corrected by the typist while typing, 0.95% were detected and corrected during proofreading by the typist, and 0.52% were not detected by the typist.
- Students performing calculator tasks and table lookup tasks: per multipart calculation, per table lookup, 1% to 2%.^[290]

16.2.8 Heuristics and biases

In the early 1970s Amos Tversky, Daniel Kahneman, and other psychologists^[210] performed studies, the results of which suggested people reason and make decisions in ways that systematically violate (mathematical based) rules of rationality. These studies covered a broad range of problems that might occur under quite ordinary circumstances. The results sparked the growth of a very influential research program often known as the *heuristics and biases* program.

There continues to be considerable debate over exactly what conclusions can be drawn from the results of these studies. Many researchers in the heuristics and biases field claim that people lack the underlying rational competence to handle a wide range of reasoning tasks, and that they exploit a collection of simple heuristics to solve problems. It is the use of these heuristics that make them prone to non-normative patterns of reasoning, or biases. This position, sometimes called the *standard picture*, claims that the appropriate norms for reasoning must be derived from mathematical logic, probability, and decision theory. An alternative to the standard Picture is proposed by evolutionary psychology. These researchers hold that logic and probability are not the norms against which human reasoning performance should be measured.

When reasoning about source code the appropriate norm is provided by the definition of the programming language used (which invariably has a basis in at least first order predicate calculus). This is not to say that probability theory is not used during software development. For instance, a developer may choose to make use of information on commonly occurring cases (such usage is likely to be limited to ordering by frequency or probability; Bayesian analysis is rarely seen).

What do the results of the heuristics and biases research have to do with software development, and do they apply to the kind of people who work in this field? The subjects used in these studies were not, at the time of the studies, software developers. Would the same results have been obtained if software developers

had been used as subjects? This question implies that developers' cognitive processes, either through training or inherent abilities, are different from those of the subjects used in these studies. The extent to which developers are susceptible to the biases, or use the heuristics, found in these studies is unknown. Your author assumes that they are guilty until proven innocent.

Another purpose for describing these studies is to help the reader get past the idea that people exclusively apply mathematical logic and probability in problem solving.

16.2.8.1 Reasoning

Comprehending source code involves performing a significant amount of reasoning over a long period of time. People generally consider themselves to be good at reasoning. However, anybody who has ever written a program knows how many errors are made. These errors are often claimed, by the author, to be caused by any one of any number of factors, except poor reasoning ability. In practice people are good at certain kinds of reasoning problems (the kind seen in everyday life) and very poor at others (the kind that occur in mathematical logic).

The basic mechanisms used by the human brain, for reasoning, have still not been sorted out and are an area of very active research. There are those who claim that the mind is some kind of general-purpose processor, while others claim that there are specialized units designed to carry out specific kinds of tasks (such as solving specific kinds of reasoning problems). Without a general-purpose model of human reasoning, there is no more to be said in this section. Specific constructs involving specific reasoning tasks are discussed in the relevant sentences.

16.2.8.2 Rationality

Many of those who study software developer behavior (there is no generic name for such people) have a belief in common with many economists. Namely, that their subjects act in a rational manner, reaching decisions for well-articulated goals using mathematical logic and probability, and making use of all the necessary information. They consider decision making that is not based on these norms as being *irrational*.

Deciding which decisions are the rational ones to make requires a norm to compare against. Many early researchers assumed that mathematical logic and probability were the norm against which human decisions should be measured. The term *bounded rationality*^[412] is used to describe an approach to problem solving performed when limited cognitive resources are available to process the available information. A growing number of studies^[143] are finding that the methods used by people to make decisions and solve problems are often optimal, given the resources available to them. A good discussion of the issues, from a psychology perspective, is provided by Samuels, Stich and Faucher.^[390]

For some time a few economists have been arguing that people do not behave according to mathematical norms, even when making decisions that will affect their financial well-being.^[285] Evidence for this heresy has been growing. If people deal with money matters in this fashion, how can their approach to software development fare any better? Your author takes the position, in selecting some of the guideline recommendations in this book, that developers' cognitive processes when reading and writing source are no different than at other times.

When reading and writing source code written in the C language, the rationality norm is defined in terms of the output from the C abstract machine. Some of these guideline recommendations are intended to help ensure that developers' comprehension of source agrees with this norm.

16.2.8.3 Risk asymmetry

The term *risk asymmetry* refers to the fact that people are *risk averse* when deciding between alternatives that have a positive outcome, but are *risk seeking* when deciding between alternatives that have a negative outcome.

Making a decision using uncertain information involves an element of risk; the decision may not be the correct one. How do people handle risk?

Kahneman and Tversky^[214] performed a study in which subjects were asked to make choices about gaining or losing money. The theory they created, *prospect theory*, differed from the accepted theory of the day,

0 developer
mental characteris-
tics

developer
reasoning

selection
statement
syntax
logical-AND-
expression
syntax
logical-OR-
expression
syntax

developer
rationality

bounded
rationality

risk asymmetry

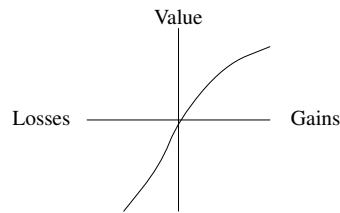


Figure 0.30: Relationship between subjective value to gains and to losses. Adapted from Kahneman.^[214]

expected utility theory (which still has followers). Subjects were presented with the following problems:

Problem 1: In addition to whatever you own, you have been given 1,000. You are now asked to choose between:

- A: Being given a further 1,000, with probability 0.5
- B: Being given a further 500, unconditionally

Problem 2: In addition to whatever you own, you have been given 2,000. You are now asked to choose between:

- C: Loosing 1,000, with probability 0.5
- D: Loosing 500, unconditionally

The majority of the subjects chose B (84%) in the first problem, and C (69%) in the second. These results, and many others like them, show that people are risk averse for positive prospects and risk seeking for negative ones (see Figure 0.30).

In the following problem the rational answer, based on knowledge of probability, is E; however, 80% of subjects chose F.

Problem 3: You are asked to choose between:

- E: Being given 4,000, with probability 0.8
- F: Being given 3,000, unconditionally

Kahneman and Tversky also showed that people's subjective probabilities did not match the objective probabilities. Subjects were given the following problems:

Problem 4: You are asked to choose between:

- G: Being given 5,000, with probability 0.001
- H: Being given 5, unconditionally

Problem 5: You are asked to choose between:

- I: Loosing 5,000, with probability 0.001
- J: Loosing 5, unconditionally

Most the subjects chose G (72%) in the first problem and J (83%) in the second.

Problem 4 could be viewed as a lottery ticket (willing to forego a small amount of money for the chance of winning a large amount), while Problem 5 could be viewed as an insurance premium (willingness to pay a small amount of money to avoid the possibility of having to pay out a large amount).

The decision weight given to low probabilities tends to be higher than that warranted by the evidence. The decision weight given to other probabilities tends to be lower than that warranted by the evidence (see Figure 0.31).

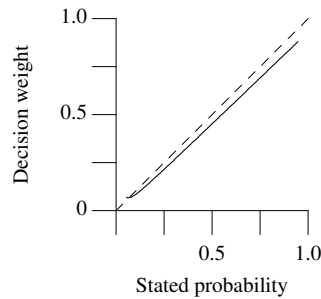


Figure 0.31: Possible relationship between subjective and objective probability. Adapted from Kahneman.^[214]

16.2.8.4 Framing effects

The framing effect occurs when alternative framings of what is essentially the same decision task cause predictably different choices.

framing effect

Kahneman and Tversky^[213] performed a study in which subjects were asked one of the following question:

Imagine that the U.S. is preparing for the outbreak of an unusual Asian disease, which is expected to kill 600 people. Two alternative programs to combat the disease have been proposed. Assume that the exact scientific estimates of the consequences of the programs are as follows:

If Program A is adopted, 200 people will be saved.

If Program B is adopted, there is a one-third probability that 600 people will be saved and a two thirds probability that no people will be saved.

Which of the two programs would you favor?

This problem is framed in terms of 600 people dying, with the option being between two programs that save lives. In this case subjects are risk averse with a clear majority, 72%, selecting Program A. For the second problem the same cover story was used, but subjects were asked to select between differently worded programs:

If Program C is adopted, 400 people will die.

If Program D is adopted, there is a one-third probability that nobody will die and two-thirds probability that 600 people will die.

In terms of their consequences Programs A and B are mathematically the same as C and D, respectively. However, this problem is framed in terms of no one dying. The best outcome would be to maintain this state of affairs. Rather than accept an unconditional loss, subjects become risk seeking with a clear majority, 78%, selecting Program D.

Even when subjects were asked both questions, separated by a few minutes, the same reversals in preference were seen. These results have been duplicated in subsequent studies by other researchers.

16.2.8.5 Context effects

The standard analysis of the decision's people make assumes that they are procedure-invariant; that is, assessing the attributes presented by different alternatives should always lead to the same one being selected. Assume, for instance, that in a decision task, a person chooses alternative X, over alternative Y. Any previous decisions they had made between alternatives similar to X and Y would not be thought to affect later decisions. Similarly, the addition of a new alternative to the list of available alternatives should not cause Y to be selected, over X.

context effects

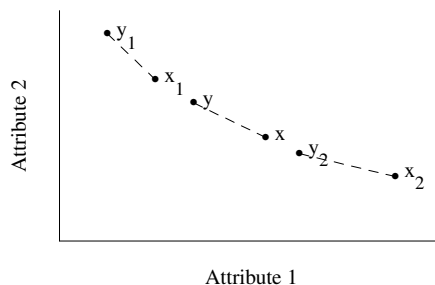


Figure 0.32: Text of background trade-off. Adapted from Tversky.^[466]

People will show procedure-invariance if they have well-defined values and strong beliefs. In these cases the appropriate values might be retrieved from a master list of preferences held in a person’s memory. If preferences are computed using some internal algorithm, each time a person has to make a decision, then it becomes possible for context to have an affect on the outcome.

Context effects have been found to occur because of the prior history of subjects answering similar questions, background context, or because of presentation of the problem itself, local context. The following two examples are taken from a study by Tversky and Simonson.^[466]

To show that prior history plays a part in a subjects judgment, Tversky and Simonson split a group of subjects in two. The first group was asked to decide between the alternatives X_1 and Y_1 , while the second group was asked to select between the options X_2 and Y_2 . Following this initial choice all subjects were asked to chose between X and Y .

Table 0.20: Percentage of each alternative selected by subject groups S_1 and S_2 . Adapted from Tversky.^[466]

Warranty	Price	S_1	S_2
X_1	\$85	12%	
Y_1	\$91	88%	
X_2	\$25		84%
Y_2	\$49		16%
X	\$60	57%	33%
Y	\$75	43%	67%

Subjects previously exposed to a decision where a small difference in price (see Table 0.20) (\$85 vs. \$91) was associated with a large difference in warranty (55,000 miles vs. 75,000 miles), were more likely to select the less-expensive tire from the target set (than those exposed to the other background choice, where a large difference in price was associated with a small difference in warranty).

In a study by Simonson and Tversky,^[414] subjects were asked to decide between two microwave ovens. Both were on sale at 35% off the regular price, at sale prices of \$109.99 and \$179.99. In this case 43% of the subjects selected the more expensive model. For the second group of subjects, a third microwave oven was added to the selection list. This third oven was priced at \$199.99, 10% off its regular price. The \$199.99 microwave appeared inferior to the \$179.99 microwave (it had been discounted down from a lower regular price by a smaller amount), but was clearly superior to the \$109.99 model. In this case 60% selected the \$179.99 microwave (13% chose the more expensive microwave). The presence of a third alternative had caused a significant number of subjects to switch the model selected.

16.2.8.6 Endowment effect

Studies have shown that losses are valued far more than gains. This asymmetry in the value assigned, by people, to goods can be seen in the endowment effect. A study performed Knetsch^[226] illustrates this effect.

Subjects were divided into three groups. The first group of was given a coffee mug, the second group

was given a candy bar, and the third group was given nothing. All subjects were then asked to complete a questionnaire. Once the questionnaires had been completed, the first group was told that they could exchange their mugs for a candy bar, the second group that they could exchange their candy bar for a mug, while the third group was told they could decide between a mug or a candy bar. The mug and the candy bar were sold in the university bookstore at similar prices.

Table 0.21: Percentage of subjects willing to exchange what they had been given for an equivalently priced item. Adapted from Knetsch.^[226]

Group	Yes	No
Give up mug to obtain candy	89%	11%
Give up candy to obtain mug	90%	10%

The decisions made by the third group, who had not been given anything before answering the questionnaire, were: mug 56%, candy 44%. This result showed that the perceived values of the mug and candy bar were close to each other.

The decisions made by the first and second groups (see Table 0.21) showed that they placed a higher value on a good they owned than one they did not own (but could obtain via a simple exchange).

The endowment effect has been duplicated in many other studies. In some studies, subjects required significantly more to sell a good they owned than they would pay to purchase it.

16.2.8.7 Representative heuristic

The representative heuristic evaluates the probability of an uncertain event, or sample, by the degree to which it

representative heuristic

- is similar in its essential attributes to the population from which it is drawn, and
- reflects the salient attributes of the process that generates it

given two events, X and Y. The event X is judged to be more probable than Y when it is more representative. The term *subjective probability* is sometimes used to describe these probabilities. They are subjective in the sense that they are created by the people making the decision. *Objective probability* is the term used to describe the values calculated from the stated assumptions, according to the axioms of mathematical probability.

Selecting alternatives based on the representativeness of only some of their attributes can lead to significant information being ignored; in particular the nonuse of base-rate information provided as part of the specification of a problem.

Treating representativeness as an operator, it is a (usually) directional relationship between a family, or process M, and some instance or event X, associated with M. It can be defined for (1) a value and a distribution, (2) an instance and a category, (3) a sample and a population, or (4) an effect and a cause. These four basic cases of representativeness occur when (Tversky^[464]):

1. M is a family and X is a value of a variable defined in this family. For instance, the representative value of the number of lines of code in a function. The most representative value might be the mean for all the functions in a program, or all the functions written by one author.
2. M is a family and X is an instance of that family. For instance, the number of lines of code in the function *foo_bar*. It is possible for an instance to be a family. The *Robin* is an instance of the bird family and a particular individual can be an instance of the Robin family.
3. M is a family and X is a subset of M. Most people would agree that the population of New York City is less representative of the US than the population of Illinois. The criteria for representativeness in

a subset is not the same as for one instance. A single instance can represent the primary attributes of a family. A subset has its own range and variability. If the variability of the subset is small, it might be regarded as a category of the family, not a subset. For instance, the selected subset of the family birds might only include Robins. In this case, the set of members is unlikely to be regarded as a representative subset of the bird family.

- 4. M is a (causal) system and X is a (possible) instance generated by it. Here M is no longer a family of objects, it is a system for generating instances. An example would be the mechanism of tossing coins to generate instances of heads and tails.

16.2.8.7.1 Belief in the law of small numbers

law of small numbers

Studies have shown that people have a strong belief in what is known as the law of small numbers. This “law” might be stated as: “Any short sequence of events derived from a random process shall have the same statistical properties as that random process.” For instance, if a fairly balanced coin is tossed an infinite number of times the percentage of heads seen will equal the percentage of tails seen. However, according to the law of small numbers, any short sequence of coin tosses will also have this property. Statistically this is not true, the sequences *HHHHHHHHHH* and *THHTHTHTH* are equally probable, but one of them does not appear to be representative of a random sequence.

Readers might like to try the following problem.

The mean IQ of the population of eighth graders in a city is *known* to be 100. You have selected a random sample of 50 children for a study of educational achievement. The first child tested has an IQ of 150.

What do you expect the mean IQ to be for the whole sample?

Did you believe that because the sample of 50 children was randomly chosen from a large population, with a known property, that it would also have this property?; that is, the answer would be 100? The effect of a child with a high IQ being canceled out by a child with a very low IQ? The correct answer is 101; the known information, from which the mean should be calculated, is that we have 49 children with an estimated average of 100 and one child with a known IQ of 150.

16.2.8.7.2 Subjective probability

subjective probability

In a study by Kahneman and Tversky,^[212] subjects were divided into two groups. Subjects in one group were asked the *more than* question, and those in the other group the *less than* question.

An investigator studying some properties of a language selected a paperback and computed the average word-length in every page of the book (i.e., the number of letters in that page divided by the number of words). Another investigator took the first line in each page and computed the line’s average word-length. The average word-length in the entire book is four. However, not every line or page has exactly that average. Some may have a higher average word-length, some lower.

The first investigator counted the number of pages that had an average word-length of 6 or (more/less) and the second investigator counted the number of lines that had an average word-length of 6 or (more/less). Which investigator do you think recorded a larger number of such units (pages for one, lines for the other)?

Table 0.22: Percentage of subjects giving each answer. Correct answers are starred. Adapted from Kahneman.^[212]

Choice	Less than 6	More than 6
The page investigator	20.8%*	16.3%
The line investigator	31.3%	42.9%*
About the same (i.e., within 5% of each other)	47.9%	40.8%

The results (see Table 0.22) showed that subjects judged equally representative outcomes to be equally likely, the size of the sample appearing to be ignored.

When dealing with samples, those containing the smaller number of members are likely to exhibit the largest variation. In the preceding case, the page investigator is using the largest sample size and is more likely to be closer to the average (4), which is less than 6. The line investigator is using a smaller sample of the book's contents and is likely to see a larger variation in measured word length (more than 6 is the correct answer here).

16.2.8.8 Anchoring

Answers to questions can be influenced by completely unrelated information. This was dramatically illustrated in a study performed by Tversky and Kahneman.^[463] They asked subjects to estimate the percentage of African countries in the United Nations. But, before stating their estimate, subjects were first shown an arbitrary number, which was determined by spinning a *wheel of fortune* in their presence. In some cases, for instance, the number 65 was selected, at other times the number 10. Once a number had been determined by the *wheel of fortune* subjects were asked to state whether the percentage of African countries in the UN was higher or lower than this number, and their estimate of the percentage. The median estimates were 45% of African countries for subjects whose *anchoring* number was 65, and 25% for subjects whose *anchoring* number was 10.

Anchoring

The implication of these results is that people's estimates can be substantially affected by a numerical *anchoring* value, even when they are aware that the anchoring number has been randomly generated.

16.2.8.9 Belief maintenance

Belief comes in various forms. There is *disbelief* (believing a statement to be false), *nonbelief* (not believing a statement to be true), *half-belief*, *quarter-belief*, and so on (the degrees of belief range from barely accepting a statement, to having complete conviction a statement is true). Knowledge could be defined as belief plus complete conviction and conclusive justification.

belief maintenance

The following are two approaches as to how beliefs might be managed.

1. The *foundation approach* argues that beliefs are derived from reasons for these beliefs. A belief is justified if and only if (1) the belief is self-evident and (2) the belief can be derived from the set of other justified beliefs (circularity is not allowed).
2. The *coherence approach* argues that where beliefs originated is of no concern. Instead, beliefs must be logically coherent with other beliefs (believed by an individual). These beliefs can mutually justify each other and circularity is allowed. A number of different types of coherence have been proposed, including *deductive coherence* (requires a logically consistent set of beliefs), *probabilistic coherence* (assigns probabilities to beliefs and applies the requirements of mathematical probability to them), *semantic coherence* (based on beliefs that have similar meanings), and *explanatory coherence* (requires that there be a consistent explanatory relationship between beliefs).

The *foundation approach* is very costly (in cognitive effort) to operate. For instance, the reasons for beliefs need to be remembered and applied when considering new beliefs. Studies^[386] show that people exhibit a belief preservation effect; they continue to hold beliefs after the original basis for those beliefs no longer

holds. The evidence suggests that people use some form of *coherence approach* for creating and maintaining their beliefs.

There are two different ways doubt about a fact can occur. When the truth of a statement is not known because of a lack of information, but the behavior in the long run is known, we have *uncertainty*. For instance, the outcome of the tossing of a coin is uncertain, but in the long run the result is known to be heads (or tails) 50% of the time. The case in which truth of a statement can never be precisely specified (indeterminacy of the average behavior) is known as *imprecision*; for instance, “it will be sunny tomorrow”. It is possible for a statement to contain both uncertainty and imprecision. For instance, the statement, “It is likely that John is a young fellow”, is uncertain (John may not be a *young fellow*) and imprecise (*young* does not specify an exact age). For a mathematical formulation, see Paskin.^[338]

Coding guidelines need to take into account that developers are unlikely to make wholesale modifications to their existing beliefs to make them consistent with any guidelines they are expected to adhere to. Learning about guidelines is a two-way process. What a developer already knows will influence how the guideline recommendations themselves will be processed, and the beliefs formed about their meaning. These beliefs will then be added to the developer’s existing personal beliefs.^[494]

16.2.8.9.1 The Belief-Adjustment model

A belief may be based on a single piece of evidence, or it may be based on many pieces of evidence. How is an existing belief modified by the introduction of new evidence? The belief-adjustment model of Hogarth and Einhorn^[170] offers an answer to this question. This subsection is based on that paper. The basic equation for this model is:

$$S_k = S_{k-1} + w_k[s(x_k) - R] \quad (0.23)$$

where: S_k is the degree of belief (a value between 0 and 1) in some hypothesis, impression, or attitude after evaluating k items of evidence; S_{k-1} is the anchor, or prior opinion (S_0 denotes the initial belief). $s(x_k)$ is the subjective evaluation of the k th item of evidence (different people may assign different values for the same evidence, x_k); R is the reference point, or background, against which the impact of the k th item of evidence is evaluated. w_k is the adjustment weight (a value between zero and one) for the k th item of evidence.

The encoding process

When presented with a statement, people can process the evidence it contains in several ways. They can use an *evaluation* process or an *estimation* process.

The *evaluation* process encodes new evidence relative to a fixed point—the hypothesis addressed by a belief. If the new evidence supports the hypothesis, a person’s belief is increased, but that belief is decreased if it does not support the hypothesis. This increase, or decrease, occurs irrespective of the current state of a person’s belief. For this case $R = 0$, and the belief-adjustment equation simplifies to:

$$S_k = S_{k-1} + w_k s(x_k) \quad (0.24)$$

where: $-1 \leq s(x_k) \leq 1$

An example of an evaluation process might be the belief that the object X always holds a value that is numerically greater than Y.

The *estimation* process encodes new evidence relative to the current state of a person’s beliefs. For this case $R = S_{k-1}$, and the belief-adjustment equation simplifies to:

$$S_k = S_{k-1} + w_k(s(x_k) - S_{k-1}) \quad (0.25)$$

where: $0 \leq s(x_k) \leq 1$

In this case the degree of belief, in a hypothesis, can be thought of as a moving average. For an estimation process, the order in which evidence is presented can be significant. While reading source code written by somebody else, a developer will form an opinion of the quality of that person's work. The judgment of each code sequence will be based on the readers current opinion (at the time of reading) of the person who wrote it.

Processing

It is possible to consider $s(x_k)$ as representing either the impact of a single piece of evidence (so-called *Step-by-Step*, SbS), or the impact of several pieces of evidence (so-called *End-of-Sequence*, EoS).

$$S_k = S_0 + w_k[s(x_1, \dots, x_k) - R] \quad (0.26)$$

where $s(x_1, \dots, x_k)$ is some function, perhaps a weighted average, of the individual subjective evaluations.

If a person is required to give a Step-by-Step response when presented with a sequence of evidence, they obviously have to process the evidence in this mode. A person who only needs to give an End-of-Sequence response can process the evidence using either SbS or EoS. The process used is likely to depend on the nature of the problem. Aggregating, using EoS, evidence from a long sequence of items of evidence, or a sequence of complex evidence, is likely to require a large amount of cognitive processing, perhaps more than is available to an individual. Breaking a task down into smaller chunks by using an SbS process, enables it to be handled by a processor having a limited cognitive capacity. Hogarth and Einhorn proposed that when people are required to provide an EoS response they use an EoS process when the sequence of items is short and simple. As the sequence gets longer, or more complex, they shift to an SbS process, to keep the peak cognitive load (of processing the evidence) within their capabilities.

Adjustment weight

The adjustment weight, w_k , will depend on the sign of the impact of the evidence, $[s(x_k) - R]$, and the current level of belief, S_k . Hogarth and Einhorn argue that when $s(x_k) \leq R$:

$$w_k = \alpha S_{k-1} \quad (0.27)$$

$$S_k = S_{k-1} + \alpha S_{k-1} s(x_k) \quad (0.28)$$

and that when $s(x_k) > R$:

$$w_k = \beta(1 - S_{k-1}) \quad (0.29)$$

$$S_k = S_{k-1} + \beta(1 - S_{k-1})s(x_k) \quad (0.30)$$

where α and β ($0 \leq \alpha, \beta \leq 1$) represent sensitivity toward positive and negative evidence. Small values indicating low sensitivity to new evidence and large values indicating high sensitivity. The values of α and β will also vary between people. For instance, some people have a tendency to give negative evidence greater weight than positive evidence. People having strong attachments to a particular point of view may not give evidence that contradicts this view any weight.^[449]

Order effects

It can be shown^[170] that use of an SbS process when $R = S_{k-1}$ leads to a recency effect. When $R = 0$, a recency effect only occurs when there is a mixture of positive and negative evidence (there is no recency effect if the evidence is all positive or all negative).

The use of an EoS process leads to a primacy effect; however, a task may not require a response until all the evidence is seen. If the evidence is complex, or there is a lot of it, people may adopt an SbS process. In this case, the effect seen will match that of an SbS process.

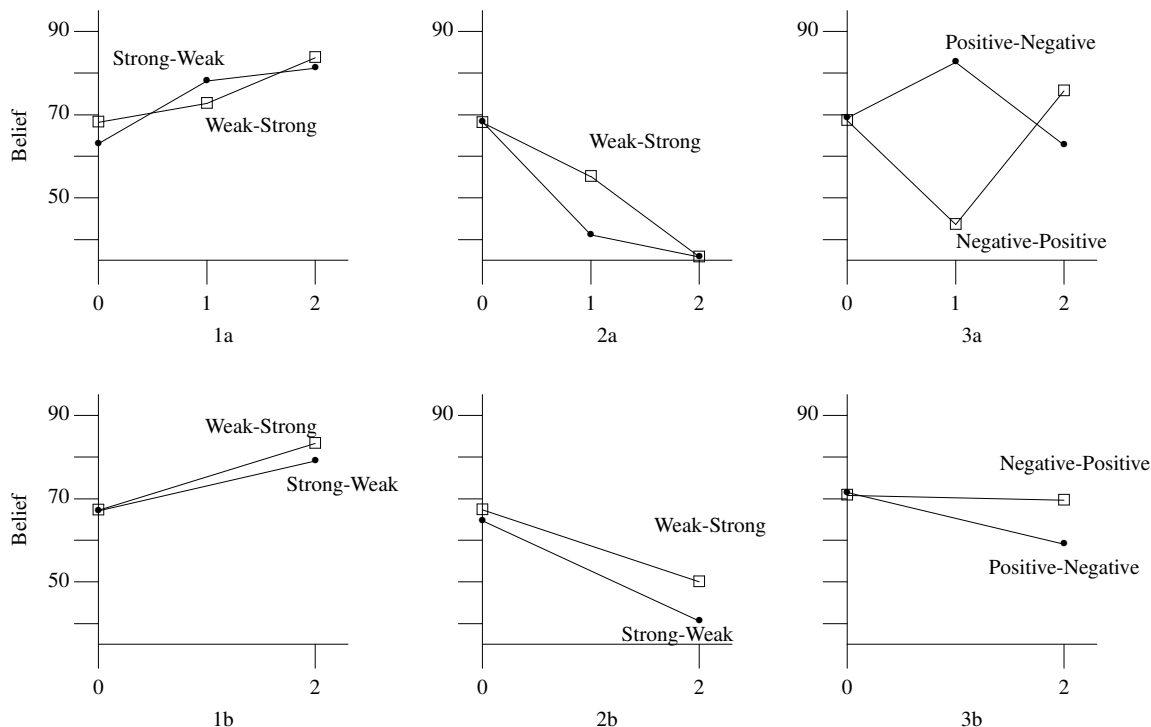


Figure 0.33: Subjects belief response curves for positive weak–strong, negative weak–strong, and positive–negative evidence; (a) Step-by-Step, (b) End-of-Sequence. Adapted from Hogarth.^[170]

recency effect
primacy effect

A *recency effect* occurs when the most recent evidence is given greater weight than earlier evidence. A *primacy effect* occurs when the initial evidence is given greater weight than later evidence.

Study

A study by Hogarth and Einhorn^[170] investigated order, and response mode, effects in belief updating. Subjects were presented with a variety of scenarios (e.g., a defective stereo speaker thought to have a bad connection, a baseball player whose hitting has improved dramatically after a new coaching program, an increase in sales of a supermarket product following an advertising campaign, the contracting of lung cancer by a worker in a chemical factory). Subjects read an initial description followed by two or more additional items of evidence. The additional evidence might be positive (e.g., “The other players on Sandy’s team did not show an unusual increase in their batting average over the last five weeks”) or negative (e.g., “The games in which Sandy showed his improvement were played against the last-place team in the league”). This positive and negative evidence was worded to create either strong or weak forms.

The evidence was presented in a variety of orders (positive or negative, weak or strong). Subjects were asked, “Now, how likely do you think X caused Y on a scale of 0 to 100?” In some cases, subjects had to respond after seeing each item of evidence: in other cases, subjects had to respond after seeing all the items.

The results (see Figure 0.33) only show a recency effect when the evidence is mixed, as predicted for the case $R = 0$.

Other studies have duplicated these results. For instance, professional auditors have been shown to display recency effects in their evaluation of the veracity of company accounts.^[339,458]

16.2.8.9.2 Effects of beliefs

The persistence of beliefs after the information they are based on has been discredited is an important issue in developer training.

Studies of physics undergraduates^[283] found that many hours of teaching only had a small effect on their

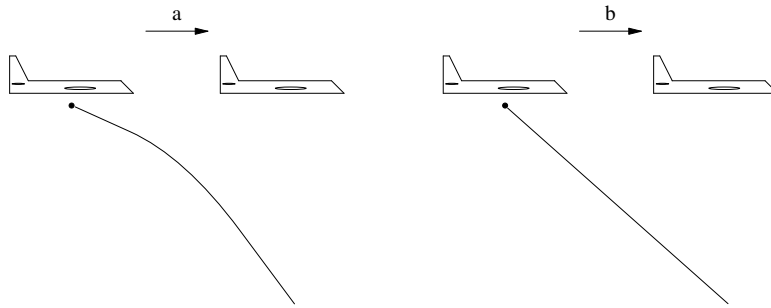


Figure 0.34: Two proposed trajectories of a ball dropped from a moving airplane. Based on McCloskey.^[283]

qualitative understanding of the concepts taught. For instance, predicting the motion of a ball dropped from an airplane (see Figure 0.34). Many students predicted that the ball would take the path shown on the right (*b*). They failed to apply what they had been taught over the years to pick the path on the left (*a*).

A study by Ploetzner and VanLehn^[349] investigated subjects who were able to correctly answer these conceptual problems. They found that the students were able to learn and apply information that was implicit in the material taught. Ploetzner and VanLehn also built a knowledge base of 39 rules needed to solve the presented problems, and 85 rules needed to generate the incorrect answers seen in an earlier study.

A study by Pazzani^[341] showed how beliefs can increase, or decrease, the amount of effort needed to deduce a concept. Two groups of subjects were shown pictures of people doing something with a balloon. The balloons varied in color (yellow or purple) and size (small or large), and the people (adults or five-year-old children) were performing some operation (stretching balloons or dipping them in water). The first group of subjects had to predict whether the picture was an “example of an *alpha*”, while the second group had to “predict whether the balloon will be inflated”. The picture was then turned over and subjects saw the answer. The set of pictures was the same for both groups of subjects.

The conditions under which the picture was an *alpha* or *inflate* were the same, a conjunctive condition (age == adult) || (action == stretching) and a disjunctive condition (size == small) && (color == yellow).

The difference between these two tasks to predict is that the first group had no prior beliefs about *alpha* situations, while it was assumed the second group had background knowledge on inflating balloons. For instance, balloons are more likely to inflate after they have been stretched, or an adult is doing the blowing rather than a child.

The other important point to note is that people usually require more effort to learn conjunctive conditions than they do to learn disjunctive conditions.

The results (see Figure 0.35) show that, for the *inflate* concept, subjects were able to make use of their existing beliefs to improve performance on the disjunctive condition, but these beliefs caused a decrease in performance on the conjunctive condition (being small and yellow is not associated with balloons being difficult to inflate).

A study by Gilbert, Tafarodi, and Malone^[144] investigated whether people could comprehend an assertion without first believing it. The results suggested that their subjects always believed an assertion presented to them, and that only once they had comprehended it were they in a position to, possibly, *unbelieve* it. The experimental setup used, involved presenting subjects with an assertion and interrupting them before they had time to *unbelieve* it. This finding has implications for program comprehension in that developers sometimes only glance at code. Ensuring that what they see does not subsequently need to be *unbelieved*, or is a partial statement that will be read the wrong way without other information being provided, can help prevent people from acquiring incorrect beliefs. The commonly heard teaching maxim of “always use correct examples, not incorrect ones” is an application of this finding.

conditionals
conjunctive/disjunctive

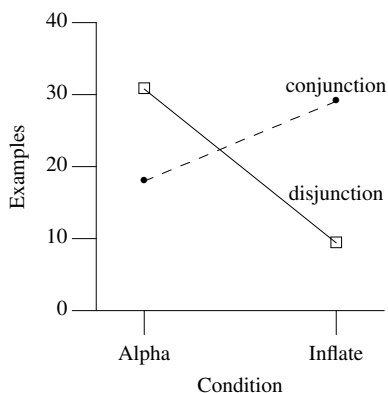


Figure 0.35: Number of examples needed before *alpha* or *inflate* condition correctly predicted in six successive pictures. Adapted from Pazzani^[341]

16.2.8.10 Confirmation bias

confirmation bias There are two slightly different definitions of the term *confirmation bias* used by psychologists, they are:

1. A person exhibits confirmation bias if they tend to interpret ambiguous evidence as (incorrectly) confirming their current beliefs about the world. For instance, developers interpreting program behavior as supporting their theory of how it operates, or using the faults exhibited by a program to conform their view that it was poorly written.
2. When asked to discover a rule that underlines some pattern (e.g., the numeric sequence 2–4–6), people nearly always apply test cases that will confirm their hypothesis. They rarely apply test cases that will falsify their hypothesis.

overcon-
fidence

Rabin and Schrag^[370] built a model showing that confirmation bias leads to overconfidence (people believing in some statement, on average, more strongly than they should). Their model assumes that when a person receives evidence that is counter to their current belief, there is a positive probability that the evidence is misinterpreted as supporting this belief. They also assume that people always correctly recognize evidence that confirms their current belief. Compared to the correct statistical method, Bayesian updating, this behavior is biased toward confirming the initial belief. Rabin and Schrag showed that, in some cases, even an infinite amount of evidence would not necessarily overcome the effects of confirmatory bias; over time a person may conclude, with near certainty, that an incorrect belief is true.

The second usage of the term *confirmation bias* applies to a study performed by Wason,^[487] which became known as the *2–4–6 Task*. In this study subjects were asked to discover a rule known to the experimenter. They were given the initial hint that the sequence 2–4–6 was an instance of this rule. Subjects had to write down sequences of numbers and show them to the experimenter who would state whether they did, or did not, conform to the rule. When they believed they knew what the rule was, subjects had to write it down and declare it to the experimenter. For instance, if they wrote down the sequences 6–8–10 and 3–5–7, and were told that these conformed to the rule, they might declare that the rule was *numbers increasing by two*. However, this was not the experimenters rule, and they had to continue generating sequences. Wason found that subjects tended to generate test cases that confirmed their hypothesis of what the rule was. Few subjects generated test cases in an attempt to disconfirm the hypothesis they had. Several subjects had a tendency to declare rules that were mathematically equivalent variations on rules they had already declared.

8 10 12: two added each time; **14 16 18:** even numbers in order of magnitude; **20 22 24:** same reason; **1 3 5:** two added to preceding number.

The rule is that by starting with any number two is added each

time to form the next number.

2 6 10: middle number is the arithmetic mean of the other two;

1 50 99: same reason.

The rule is that the middle number is the arithmetic mean of the other two.

3 10 17: same number, seven, added each time; 0 3 6;
three added each time.

The rule is that the difference between two numbers next to each other is the same.

12 8 4: the same number subtracted each time to form the next number.

The rule is adding a number, always the same one to form the next number.

1 4 9: any three numbers in order of magnitude.

The rule is any three numbers in order of magnitude.

Sample 2-4-6 subject protocol. Adapted from Wason.^[487]

The actual rule used by the experimenter was “three numbers in increasing order of magnitude”.

These findings have been duplicated in other studies. In a study by Mynatt, Doherty, and Tweney,^[316] subjects were divided into three groups. The subjects in one group were instructed to use a confirmatory strategy, another group to use a disconfirmatory strategy, and a control group was not told to use any strategy. Subjects had to deduce the physical characteristics of a system, composed of circles and triangles, by firing particles at it (the particles, circles and triangles, appeared on a computer screen). The subjects were initially told that “triangles deflect particles”. In 71% of cases subjects selected confirmation strategies. The instructions on which strategy to use did not have any significant effect.

In a critique of the interpretation commonly given for the results from the 2–4–6 Task, Klayman and Ha^[223] pointed out that it had a particular characteristic. The hypothesis that subjects commonly generate (*numbers increasing by two*) from the initial hint is completely contained within the experimenters rule, case 2 in Figure 0.36. Had the experimenters rule been *even numbers increasing by two*, the situation would have been that of case 3 in Figure 0.36.

Given the five possible relationships between hypothesis and rule, Klayman and Hu analyzed the possible strategies in an attempt to find one that was optimal for all cases. They found that the optimal strategy was a function of a variety of task variables, such as the base rates of the target phenomenon and the hypothesized conditions. They also proposed that people do not exhibit confirmation bias, rather people have a general all-purpose heuristic, the *positive test strategy*, which is applied across a broad range of hypothesis-testing tasks.

A *positive test strategy* tests a hypothesis by examining instances in which the property or event is expected to occur to see if it does occur. The analysis by Klayman and Hu showed that this strategy performs well in real-world problems. When the target phenomenon is relatively rare, it is better to test where it occurs (or where it was known to occur in the past) rather than where it is not likely to occur.

A study by Mynatt, Doherty, and Dragan^[315] suggested that capacity limitations of working memory were also an issue. Subjects did not have the capacity to hold information on more than two alternatives in working memory at the same time. The results of their study also highlighted the fact that subjects process the alternatives in *action* (what to do) problems differently than in *inference* (what is) problems.

Karl Popper^[353] pointed out that scientific theories could never be shown to be logically true by generalizing from confirming instances. It was the job of scientists to try to perform experiments that attempted to falsify a theory. Popper’s work on how a hypothesis should be validated has become the generally accepted way of measuring performance (even if many scientists don’t appear to use this approach).

The fact that people don’t follow the hypothesis-testing strategy recommended by Popper is seen, by some, as a deficiency in peoples thinking processes. The theoretical work by Klayman and Hu shows that it might

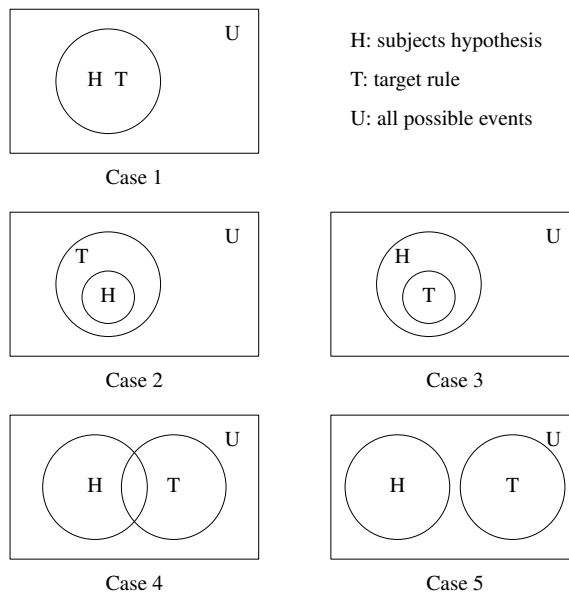


Figure 0.36: Possible relationships between hypothesis and rule. Adapted from Klayman.^[223]

be Poppers theories that are deficient. There is also empirical evidence showing that using disconfirmation does not necessarily improve performance on a deduction task. A study by Tweney, Doherty, Worner, Pliske, Mynatt, Gross, and Arkkelin^[467] showed that subjects could be trained to use a disconfirmation strategy when solving the 2–4–6 Task. However, the results showed that using this approach did not improve performance (over those subjects using a confirmation strategy).

Do developers show a preference toward using positive test strategies during program comprehension? What test strategy is the best approach during program comprehension? The only experimental work that has addressed this issue used students in various stages of their academic study. A study by Teasley, Leventhal, Mynatt, and Rohlman^[444] asked student subjects to test a program (based on its specification). The results showed that the more experienced subjects created a greater number of negative tests.

16.2.8.11 Age-related reasoning ability

It might be thought that reasoning ability declines with age, along with the other faculties. A study by Tentori, Osherson, Hasher, and May^[446] showed the opposite effect; some kinds of reasoning ability improving with age.

Consider the case of a person who has to decide between two alternatives, A and B (e.g., vanilla and strawberry ice cream), and chooses A. Adding a third alternative, C (e.g., chocolate ice cream) might entice that person to select C. A mathematical analysis shows that adding alternative C would not cause a change of preference to B. How could adding the alternative chocolate ice cream possibly cause a person who previously selected vanilla to now choose strawberry?

So-called *irregular choices* have been demonstrated in several studies. Such irregular choices seem to occur among younger (18–25) subjects, older (60–75) subjects tending to be uninfluenced by the addition of a third alternative.

16.3 Personality

To what extent does personality affect developers’ performance, and do any personality differences need to be reflected in coding guidelines?

- A study by Turley and Bieman^[460] looked for differences in the competencies of exceptional and non-exceptional developers. They found the personal attributes that differentiated performances were:

reasoning ability
age-related

developer
personality

desire to contribute, perseverance, maintenance of a *big picture* view, desire to do/bias for action, driven by a sense of mission, exhibition and articulation of strong convictions, and proactive role with management. Interesting findings in another context, but of no obvious relevance to these coding guidelines. Turley and Bieman also performed a Myers-Briggs Type Indicator (MBTI) test^[314] on their subjects. The classification containing the most developers (7 out of 20) was INTJ (*Introvert, Intuitive, Thinking, Judging*), a type that occurs in only 10% of male college graduates. In 15 out of 20 cases, the type included *Introvert, Thinking*. There was no significance in scores between exceptional and nonexceptional performers. These findings are too broad to be of any obvious relevance to coding guidelines.

Myers-Briggs
Type Indicator

- A study of the relationship between personality traits and achievement in an introductory Fortran course was made by Kagan and Douthat.^[209] They found that relatively introverted students, who were hard-driving and ambitious, obtained higher grades than their more extroverted, easy-going compatriots. This difference became more pronounced as the course progressed and became more difficult. Again these findings are too broad to be of any obvious relevance to these coding guidelines.

These personality findings do not mean that to be a good developer a person has to fall within these categories, only that many of those tested did.

It might be assumed that personality could affect whether a person enjoys doing software development, and that somebody who enjoys their work is likely to do a better job (but does personal enjoyment affect quality, or quantity of work performed?). These issues are considered to be outside the scope of this book (they are discussed a little more in Staffing,).

0 coding
guidelines
staffing

Developers are sometimes said to be paranoid. One study^[493] has failed to find any evidence for this claim.

Usage

17 Introduction

This subsection provides some background on the information appearing in the Usage subsections of this book. The purpose of this usage information is two-fold:

Usage
1
Usage
introduction

1. To give readers a feel for the common developer usage of C language constructs. Part of the process of becoming an experienced developers involves learning about what is common and what is uncommon. However, individual experiences can be specific to one application domain, or company cultures.
2. To provide frequency-of-occurrence information that could be used as one of the inputs to cost/benefit decisions (i.e., should a guideline recommendation be made rather than what recommendation might be made). This is something of a chicken-and-egg situation in that knowing what measurements to make requires having potential guideline recommendations in mind, and the results of measurements may suggest guideline recommendations (i.e., some construct occurs frequently).

0 guideline
recom-
mendations
selecting

Almost all published measurements on C usage are an adjunct to a discussion of some translator optimization technique. They are intended to show that the optimization, which is the subject of the paper, is worthwhile because some constructs occurs sufficiently often for an optimization to make worthwhile savings, or that some special cases can be ignored because they rarely occur. These kinds of measurements are usually discussed in the *Common implementation* subsections. One common difference between the measurements in Common Implementation subsections and those in Usage subsections is that the former are often dynamic (instruction counts from executing programs), while the latter are often static (counts based on some representation of the source code).

There have been a few studies whose aim has been to provide a picture of the kinds of C constructs that commonly occur (e.g., preprocessor usage,^[115] embedded systems^[111]). These studies are quoted in the relevant C sentences. There have also been a number of studies of source code usage for other algorithmic

languages, Assembler,^[85] Fortran,^[227] PL/1,^[109] Cobol^[6, 69, 199] (measurements involving nonalgorithmic languages have very different interests^[63, 75]). These are of interest in studying cross-language usage, but they are not discussed in this book. In some cases a small number of machine code instruction sequences (which might be called idioms) have been found to account for a significant percentage of the instructions executed during program execution.^[422]

The intent here is to provide a broad brush picture. On the whole, single numbers are given for the number of occurrences of a construct. In most cases there is no break down by percentage of functions, source files, programs, application domain, or developer. There is variation across all of these (e.g., application domain and individual developer). Whenever this variation might be significant, additional information is given. Those interested in more detailed information might like to make their own measurements.

Many of the coding guideline recommendations made in this book apply to the visible source code as seen by the developer. For these cases any usage measurements also apply to the visible source code. The effects of any macro replacement, conditional inclusion, or **#included** header are ignored. Each usage subsection specifies what the quoted numbers apply to (usually either visible source, or the tokens processed during translation phase 7).

In practice many applications do not execute in isolation; there is usually some form of operating system that is running concurrently with it. The design of processor instruction sets often takes task-switching and other program execution management tasks into account. In practice the dynamic profile of instructions executed by a processor reflects this mix of usage,^[43] as does the contents of its cache.^[281]

17.1 Characteristics of the source code

All source code may appear to look the same to the casual observer. An experienced developer will be aware of recurring patterns; source can be said to have a style. Several influences can affect the characteristics of source code, including the following:

- *Use of extensions to the C language and differences, for prestandard C, from the standard (often known as K&R C).* Some extensions eventually may be incorporated into a revised version of the standard; for instance, **long long** was added in C99. Some extensions are specific to the processor on which the translated program is to execute.
- *The application domain.* For instance, scientific and engineering applications tend to make extensive use of arrays and spend a large amount of their time in loops processing information held in these arrays; screen based interactive applications often contain many calls to GUI library functions and can spend more time in these functions than the developer's code; data-mining applications can spend a significant amount of time searching large data structures.
- *How the application is structured.* Some applications consist of a single, monolithic, program, while others are built from a collection of smaller programs sharing data with one another. These kinds of organization affect how types and objects are defined and used.
- *The extent to which the source has evolved over time.* Developers often adopt the low-risk strategy of making the minimal number of changes to a program when modifying it. Often this means that functions and sequences of related statements tend to grow much larger than would be the case if they had been written from scratch, because no restructuring is performed.
- *Individual or development group stylistic usage.* These differences can include the use of large or small functions, the use of enumeration constants or object-like macros, the use of the smallest integer type required rather than always using **int**, and so forth.

17.2 What source code to measure?

This book is aimed at a particular audience and the source code they are likely to be actively working on. This audience will be working on C source that has been written by more than one developer, has existed for a year or more, and is expected to continue to be worked on over the coming years.

The benchmarks used in various application areas were written with design aims that differ from those of [this book](#). For instance, the design aim behind the choice of programs in the SPEC CPU benchmark suite was to measure processor, memory hierarchy, and translator performance. Many of these programs were written by individuals, are relatively short, and have not changed much over time.

Although there is a plentiful supply of C source code publicly available (an estimated 20.3 million C source files on the Web^[45]), this source is nonrepresentative in a number of ways, including:

- The source has had many of the original defects removed from it. The ideal time to make these measurements is while the source is being actively developed.
- Software for embedded systems is often so specialized (in the sense of being tied to custom hardware), or commercially valuable, that significant amounts of it are not usually made publicly available.

Nevertheless, a collection of programs was selected for measurement, and the results are included in this book (see [Table 0.23](#)). The programs used for this set of measurements have reached the stage that somebody has decided that they are worth releasing. This means that some defects in the source, prior to the release, will not be available to be included in these usage figures.

Table 0.23: Programs whose source code (i.e., the .c and .h files) was used as the input to measurement tools (operating on either the visible or translated forms), whose output was used to generate this book's usage figures and tables.

Name	Application Domain	Version
gcc	C compiler	2.95
idsoftware	Games programs, e.g., Doom	
linux	Operating system	2.4.20
mozilla	Web browser	1.0
openafs	File system	1.2.2a
openMotif	Window manager	2.2.2
postgresql	Database system	6.5.3

Table 0.24: Source files excluded from the Usage measurements.

Files	Reason for Exclusion
gcc-2.95/libio/tests/tfformat.c	a list of approximately 4,000 floating constants
gcc-2.95/libio/tests/tiformat.c	a list of approximately 5,000 hexadecimal constants

Table 0.25: Character sequences used to denote those operators and punctuators that perform more than one role in the syntax.

Symbol	Meaning	Symbol	Meaning
++v	prefix ++	--v	prefix --
v++	postfix ++	v--	postfix --
-v	unary minus	+v	unary plus
*v	indirection operator	*p	star in pointer declaration
&v	address-of		
:b	colon in bitfield declaration	?:	colon in ternary operator

17.3 How were the measurements made?

The measurements were based two possible interpretations of the source (both of them static, that is, based on the source code, not program execution):

- *The visible source.* This is the source as it might be viewed in a source code editor. The quoted results specify whether the .c or the .h files, or both, were used. The tools used to make these measurements

are based on either analyzing sequences of characters or sequences of preprocessing tokens (built from the sequences of characters). The source of the tools used to make these measurements is available on this book's Web site: <http://www.knosof.co.uk/cbook/cbook.html>.

- *The translated source.* This is the source as processed by a translator following the syntax and semantics of the C language. Measurements based on the translated source differ from those based on the visible source in that they may not include source occurring within some arms of conditional inclusion directives, may be affected by macro replacement, may not include all source files in the distribution (because the make-file does not require them to be translated), and do not include a few files which could not be successfully translated by the tool used. (The tools used to measure the translated source were based on a C static analysis tool.^[203]) Every attempt was made to exclude the contents of any **#included** system headers (i.e., any header using the < > delimited form) from the measurements. However, the host on which the measurements were made (RedHat 9, a Linux distribution) will have some effect; for instance, use of a macro defined in an implementation's header may expand to a variety of different preprocessing tokens, depending on the implementation. Also some application code contains conditional inclusion directives that check properties of the host O/S.

conditional
inclusion
macro re-
placement

Note. The condition for inclusion in a table containing *Common token pairs involving* information was that percentage occurrence of both tokens be greater than 1% and that the sum of both token frequencies be greater than 5%. In some cases the second requirement excluded tokens pairs when the percentage occurrence of one of the tokens was relatively high. For instance, the token pair - *character-constant* does not appear in Table ?? because the sum of the token frequencies is 4.1 (i.e., 1.9+2.2).

The usage information often included constructs that rarely occurred. Unless stated otherwise a cut-off of 1% was used. Values for table entries such as *other-types* were created by summing the usage information below this cut-off value.

References

1. P. L. Ackerman and E. D. Heggstad. Intelligence, personality, and interests: Evidence for overlapping traits. *Psychological Bulletin*, 121(2):219–245, 1997.
2. E. N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.
3. V. Agarwal, M. S. Hrishikesh, S. W. K. Doug, and Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.
4. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, 1985.
5. A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In M. P. Atkinson, M. E. Orłowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *Proceedings of the Twenty-fifth International Conference on Very Large Databases*, pages 266–277, Los Altos, CA 94022, USA, Sept. 1999. Morgan Kaufmann Publishers.
6. M. M. Al-Jarrah and I. S. Torsun. An empirical analysis of COBOL programs. *Software–Practice and Experience*, 9:341–359, 1979.
7. E. M. Altmann. Near-term memory in programming: A simulation-based analysis. *International Journal of Human-Computer Studies*, 54:189–210, 2001.
8. E. M. Altmann. Functional decay of memory for tasks. *Psychological Research*, 66(4):287–297, 2002.
9. AMD. *Software Optimization Guide for AMD Athlon 64 and AMD Opteron Processors*. Advanced Micro Devices, Inc, 3.03 edition, Sept. 2002.
10. J. R. Anderson. Interference: The relationship between response latency and response accuracy. *Journal of Experimental Psychology: Human Learning and Memory*, 7(5):326–343, 1981.
11. J. R. Anderson. *Cognitive Psychology and its Implications*. Worth Publishers, fifth edition, 2000.
12. J. R. Anderson. *Learning and Memory*. John Wiley & Sons, Inc, second edition, 2000.
13. J. R. Anderson and C. Libiere. *The Atomic Components of Thought*. Lawrence Erlbaum Associates, 1998.
14. J. R. Anderson and R. Milson. Human memory: An adaptive perspective. *Psychological Review*, 96(4):703–719, 1989.
15. Anon. Tendra home page. www.tendra.org, 2003.
16. Anon. Top 500 supercomputer sites. www.top500.org, 2003.
17. Anon. Trimaran home page. www.trimaran.org, 2003.
18. A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
19. H. R. Arkes, R. M. Dawes, and C. Christensen. Factors influencing the use of a decision rule in a probabilistic task. *Organizational Behavior and Human Decision Processes*, 37:93–110, 1986.
20. B. Armstrong and R. Eigenmann. Performance forecasting: Characterization of applications on current and future architectures. Technical Report ECE-HPCLab-97202, Purdue University School of ECE, Jan. 1997.
21. J. Backus. The history of FORTRAN I, II, and III. *SIGPLAN Notices*, 13(8):165–180, 1978.
22. D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. Technical Report UCB/CSD-93-781, University of California, Berkeley, 1993.
23. A. Baddeley. Working memory: Looking back and looking forward. *Nature Reviews*, 4(10):829–839, Oct. 2003.
24. A. Baddeley, M. Conway, and J. Aggleton. *Episodic Memory: New Directions in Research*. Oxford University Press, 2002.
25. A. D. Baddeley. *Essentials of Human Memory*. Psychology Press, 1999.
26. A. D. Baddeley, N. Thomson, and M. Buchanan. Word length and the structure of short-term memory. *Journal of Verbal Learning and Verbal Behavior*, 14:575–589, 1975.
27. I. Bahar, B. Calder, and D. Grunwald. A comparison of software code reordering and victim buffers. *ACM SIGARCH Computer Architecture News*, Mar. 1999.
28. H. P. Bahrnick. Semantic memory content in permastore: Fifty years of memory for Spanish learned in school. *Journal of Experimental Psychology: General*, 113(1):1–26, 1984.
29. J. N. Bailenson, M. S. Shum, and J. D. Coley. A bird’s eye view: Biological categorization and reasoning within and across cultures. *Cognition*, 84:1–53, 2002.
30. J. L. Bailey and G. Stefaniak. Industry perceptions of the knowledge, skills, and abilities needed by computer programmers. In *Proceedings of the 2001 ACM SIGCPR Conference on Computer Personnel Research (SIGCPR 2001)*, pages 93–99. ACM Press, 2001.
31. R. D. Banker and S. A. Slaughter. The moderating effects of structure on volatility and complexity in software enhancement. *Information Systems Research*, 11(3):219–240, Sept. 2000.
32. S. Bansal and A. Aiken. Automatic generation of peephole super-optimizers. In *Proceedings of the 12th International conference on Architectural support for programming languages and operating systems*, pages 394–403, Apr. 2006.
33. P. Banyard and N. Hunt. Something missing? *The Psychologist*, 13(2):68–71, 2000.
34. J. H. Barkow, L. Cosmides, and J. Tooby. *The Adapted Mind: Evolutionary Psychology and the Generation of Culture*. Oxford University Press, 1992.
35. V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørungård, and M. V. Zelkowitz. The empirical investigation of perspective-based reading. *Empirical Software Engineering: An International Journal*, 1(2):133–164, 1996.
36. K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. The performance and energy consumption of embedded real-time operating systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems CASES’01*, pages 203–210. ACM Press, Nov. 2001.
37. L. R. Beach, V. E. Barnes, and J. J. J. Christensen-Szalanski. Beyond heuristics and biases: A contingency model of judgmental forecasting. *Journal of Forecasting*, 5:143–157, 1986.
38. B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, second edition, 1990.
39. G. Bell and J. Gray. High performance computing: Crays, clusters, and centers. what next? Technical Report MSR-TR-2001-76, Microsoft Research, Sept. 2001.
40. M. E. Benitez and J. W. Davidson. Target-specific global code improvement: Principles and applications. Technical Report Technical Report CS-94-42, University of Virginia, 1994.
41. Y. Benkler. Coase’s penguin, or, Linux and the nature of the firm. *The Yale Law Journal*, 112(3), Dec. 2002.
42. D. C. Berry and D. E. Broadbent. On the relationship between task performance and associated verbalized knowledge. *Quarterly Journal of Experimental Psychology*, 36A:209–231, 1984.

43. R. Bhargava, J. Rubio, S. Kannan, and L. K. John. Understanding the impact of X86/NT computing on microarchitecture. In L. K. John and A. M. G. Maynard, editors, *Characterization of Contemporary Workloads*, chapter 10, pages 203–228. Kluwer Academic Publishers, 2001.
44. S. S. Bhattacharyya, R. Leupers, and P. Marwedel. Software synthesis and code generation for signal processing systems. Technical Report CS-TR-4063, University of Maryland, College Park, Sept. 1999.
45. J. M. Bieman and V. Murdock. Finding code on the world wide web: A preliminary investigation. In *Proceedings First International Workshop on Source Code Analysis and Manipulation (SCAM2001)*, pages 73–78, 2001.
46. S. Bikhchandani, D. Hirshleifer, and I. Welch. A theory of fads, fashion, custom, and cultural change as information cascades. *Journal of Political Economy*, 100(5):992–1026, 1992.
47. S. Blackmore. *The Meme Machine*. Oxford University Press, 1999.
48. T. Bonk and U. Rüdte. Performance analysis and optimization of numerically intensive programs. Technical Report SFB Bericht 342/26/92 A, Technische Universität München, Germany, Nov. 1992.
49. G. H. Bower, M. C. Clark, A. M. Lesgold, and D. Winzenz. Hierarchical retrieval schemes in recall of categorized word lists. *Journal of Verbal Learning and Verbal Behavior*, 8:323–343, 1969.
50. M. G. Bradac, D. E. Perry, and L. G. Votta. Prototyping A process monitoring experiment. *IEEE Transactions on Software Engineering*, 20(10):774–784, 1994.
51. G. L. Bradshaw and J. R. Anderson. Elaborative encoding as an explanation of levels of processing. *Journal of Verbal Learning and Verbal Behavior*, 21:165–174, 1982.
52. R. A. Brealey and S. C. Myers. *Principles of Corporate Finance*. Irwin McGraw-Hill, 2000.
53. E. J. Breen. *Extensible Interactive C*. eic.sourceforge.net, June 2000.
54. A. Brooks, J. Daly, J. Miller, M. Roper, and M. Wood. Replication of experimental results in software engineering. Technical Report ISERN-96-10, Department of Computer Science, University of Strathclyde, Livingstone Tower, Richmond Street, Glasgow G1 1XH, UK, 1996.
55. T. Budd. *An APL Compiler*. Springer-Verlag, 1988.
56. D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In *23rd Annual International Symposium on Computer Architecture*, pages 78–89, 1996.
57. Q. L. Burrell. A note on ageing in a library circulation model. *Journal of Documentation*, 41(2):100–115, 1985.
58. M. D. Byrne and S. Bovair. A working memory model of a common procedural error. *Cognitive Science*, 21(1):31–61, 1997.
59. C. F. Camerer and E. F. Johnson. The process-performance paradox in expert judgment: How can the experts know so much and predict so badly? In K. A. Ericsson and J. Smith, editors, *Towards a general theory of expertise: Prospects and limits*. Cambridge University Press, 1991.
60. M. C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, June 1996.
61. R. E. Carlson, B. H. Khoo, R. G. Yaure, and W. Schneider. Acquisition of a problem-solving skill: Levels of organization and use of working memory. *Journal of Experimental Psychology: General*, 119(2):193–214, 1990.
62. E. Carmel and S. Becker. A process model for packaged software development. *IEEE Transactions on Engineering Management*, 41(5):50–61, 1995.
63. K. A. Cassell. Tools for the analysis of large PROLOG programs. Thesis (m.s.), University of Texas at Austin, Austin, TX, 1985.
64. R. G. G. Cattell. Automatic derivation of code generators from machine descriptors. *ACM Transactions on Programming Languages and Systems*, 2(2):173–190, 1980.
65. J. P. Cavanagh. Relation between the immediate memory span and the memory search rate. *Psychological Review*, 79(6):525–530, 1972.
66. W. G. Chase and K. A. Ericsson. Skill and working memory. In G. H. Bower, editor, *The Psychology of Learning and Motivation*, pages 1–58. Academic, 1982.
67. J. B. Chen. *The Impact of Software Structure and Policy on CPU and Memory System Performance*. PhD thesis, Carnegie Mellon University, May 1994.
68. P. Cheng, K. J. Holyoak, R. E. Nisbett, and L. M. Oliver. Pragmatic versus syntactic approaches to training deductive reasoning. *Cognitive Psychology*, 18:293–328, 1986.
69. R. J. Chevanec and T. Heidet. Static profile and dynamic behavior of COBOL programs. *SIGPLAN Notices*, 13(4):44–57, Apr. 1978.
70. T. M. Chilimbi. On the stability of temporal data reference profiles. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 151–162, Sept. 2001.
71. R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong. Orthogonal defect classification – A concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, Nov. 1992.
72. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles, SOSP'01*, pages 73–88, 2001.
73. S. Chulani, B. Boehm, and B. Steece. Calibrating software cost models using bayesian analysis. Technical Report USC-CSE-98-508, American Science Institute of Technology, 1998.
74. M. Ciolkowski, C. Differding, O. Laitenberger, and J. Munch. Empirical investigation of perspective-based reading: a replicated experiment. Technical Report Technical Report ISERN-97-13, Fraunhofer Institute for Experimental Software Engineering, University of Kaiserlautern: Kaiserlautern, 1997.
75. D. W. Clark and C. C. Green. An empirical study of list structure in Lisp. *Communications of the ACM*, 20(2):78–87, Feb. 1977.
76. R. F. Cmelik, S. I. Kong, D. R. Ditzel, and E. J. Kelly. An analysis of MIPS and SPARC instruction set utilization on the SPEC benchmarks. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 290–301, 1991.
77. R. Cohn and P. G. Lowney. Design and analysis of profile-based optimization in Compaq’s compilation tools for alpha. *Journal of Instruction-Level Parallelism*, 3:1–25, 2000.
78. A. M. Collins and M. R. Quillian. Retrieval time from semantic memory. *Journal of Verbal Learning and Verbal Behavior*, 8:240–247, 1969.
79. R. Colom, I. Rebollo, A. Palacios, M. Juan-Espinosa, and P. C. Kyllonen. Working memory is (almost) perfectly predicted by g. *Intelligence*, 32:277–296, 2004.

80. S. P. Consortium. Ada 95 quality and style guide: Guidelines for professional programmers. Technical Report SPC-94093-CMC Version 01.00.10, Software Productivity Consortium, Oct. 1995.
81. D. Conway. *Perl Best Practices*. O'Reilly, 2005.
82. J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1991.
83. L. Cosmides and J. Tooby. Evolutionary psychology: A primer. Technical report, Center for Evolutionary Psychology, University of California, Santa Barbara, 1998.
84. J. D. Couger and M. A. Colter. *Maintenance Programming: Improving Productivity Through Motivation*. Prentice-Hall, Inc, 1985.
85. N. S. Coulter and N. H. Kelly. Computer instruction set usage by programmers: An empirical investigation. *Communications of the ACM*, 29(7):643–647, July 1986.
86. M. A. Covington. Some coding guidelines for Prolog. www.ai.uga.edu/mc, 2001.
87. N. Cowan. *Attention and Memory: An Integrated Framework*. Oxford University Press, 1997.
88. N. Cowan. The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24(1):87–185, 2001.
89. E. R. F. W. Crossman. A theory of the acquisition of speed-skill. *Ergonomics*, 2:153–166, 1959.
90. M. Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Harper Perennial, 1990.
91. J. Darley and C. D. Batson. From Jerusalem to Jericho: A study of situational and dispositional variables in helping behavior. *Journal of Personality and Social Psychology*, 27(1):100–108, 1973.
92. R. Dattero and S. D. Galup. Programming languages and gender. *Communications of the ACM*, 47(1):99–102, Jan. 2004.
93. T. H. Davenport and J. C. Beck. *The Attention Economy*. Harvard Business School Press, 2001.
94. J. W. Davidson, J. R. Rabung, and D. B. Whalley. Relating static and dynamic machine code measurements. Technical Report CS-89-03, Department of Computer Science, University of Virginia, July 13 1989.
95. J. W. Davison, D. M. Mand, and W. F. Opdyke. Understanding and addressing the essential costs of evolving systems. *Bell Labs Technical Journal*, 5(2):44–54, Apr.-June 2000.
96. G. C. S. de Araújo. *Code Generation Algorithms for Digital Signal Processors*. PhD thesis, Princeton University, 1997.
97. I. J. Deary and C. Stough. Intelligence and inspection time. *American Psychologist*, 51(6):599–608, 1996.
98. A. Degani and E. L. Wiener. On the design of flight-deck procedures. Technical Report 177642, NASA Ames Research Center, June 1994.
99. B. Demoen and G. Maris. A comparison of some schemes for translating logic to C. In *ICLP Workshop: Parallel and Data Parallel Execution of Logic Programs*, pages 79–91, 1994.
100. H. G. Dietz and T. I. Mattox. Compiler optimizations using data compression to decrease address reference entropy. In *LCPC '02: 15th Workshop on Languages and Compilers for Parallel Computing*, July 2002.
101. D. K. Dirlam. Most efficient chunk sizes. *Cognitive Psychology*, 3:355–359, 1972.
102. K. M. Dixit. Overview of the SPEC benchmarks. In J. Gray, editor, *The Benchmark Handbook*, chapter 9, pages 489–521. Morgan Kaufmann Publishers, 1993.
103. T. Dybå, V. B. Kampenes, and D. I. K. Sjøberg. A systematic review of statistical power in software engineering experiments. *Information and Software Technology*, 48(8):745–755, 2006.
104. H. Ebbinghaus. *Memory: A contribution to experimental psychology*. Dover Publications, 1987.
105. L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. How input data sets change program behaviour. In *Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW'02)*, Feb. 2002.
106. S. G. Eick, T. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.
107. H. J. Einhorn. Accepting error to make less error. *Journal of Personality Assessment*, 50:387–395, 1986.
108. N. C. Ellis and R. A. Hennelly. A bilingual word-length effect: Implications for intelligence testing and the relative ease of mental calculation in Welsh and English. *British Journal of Psychology*, 71:43–51, 1980.
109. J. L. Elshoff. A numerical profile of commercial PL/I programs. *Software-Practice and Experience*, 6:505–525, 1976.
110. K. E. Emam and I. Wiczorek. The repeatability of code defect classifications. Technical Report Technical Report ISERN-98-09, Fraunhofer Institute for Experimental Software Engineering, 1998.
111. J. Engblom. Why SpecInt95 should not be used to benchmark embedded systems tools. *ACM SIGPLAN Notices*, 34(7):96–103, July 1999.
112. K. A. Ericsson and N. Charness. Expert performance. *American Psychologist*, 49(8):725–747, 1994.
113. K. A. Ericsson, R. T. Krampe, and C. Tesch-Romer. The role of deliberate practice in the acquisition of expert performance. *Psychological Review*, 100:363–406, 1993. also University of Colorado, Technical Report #91-06.
114. K. A. Ericsson and A. C. Lehmann. Expert and exceptional performance: Evidence of maximal adaption to task constraints. *Annual Review of Psychology*, 47:273–305, 1996.
115. M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.
116. W. K. Estes. *Classification and Cognition*. Oxford University Press, 1994.
117. K. Ewusi-Mensah and Z. H. Przasnyski. On information systems project abandonment: An exploratory study of organizational practices. *MIS Quarterly*, 15(1):67–86, Mar. 1991.
118. M. W. Eysenck and M. T. Keane. *Cognitive Psychology: A Student's Handbook*. Psychology Press, fourth edition, 2000.
119. C. Fagot. *Chronometric investigations of task switching*. PhD thesis, University of California, San Diego, 1994.
120. R. Falk and C. Konold. Making sense of randomness: Implicit encoding as a basis for judgment. *Psychological Review*, 104(2):301–318, 1997.
121. R. J. Fateman. Software fault prevention by language choice: Why C is not my favorite language. University of California at Berkeley, 1999.

122. J. M. Favaro. Value based software reuse investment. *Annals of Software Engineering*, 5:5–52, 1998.
123. J. Feldman. Minimization of boolean complexity in human concept learning. *Nature*, 407:630–633, Oct. 2000.
124. N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(3):675–689, 1999.
125. N. E. Fenton and M. Neil. Software metrics: Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 357–370. ACM Press, 2000.
126. N. E. Fenton and S. L. Pfleeger. *Software Metrics*. PWS Publishing Company, second edition, 1997.
127. S. Feuerstein. *Oracle PL/SQL Best Practices*. O’Reilly, 2001.
128. K. Fiedler. The dependence of the conjunction fallacy on subtle linguistic factors. *Psychological Research*, 50:123–129, 1988.
129. G. J. Fitzsimons and B. Shiv. Non-conscious and contaminative effects of hypothetical questions on subsequent decision making. *Journal of Consumer Research*, 28:224–238, Sept. 2001.
130. B. Foote and J. Yoder. Big ball of mud. In *Fourth Conference on Pattern Languages of Programs (PLoP) 1997*, 1997.
131. W. B. Frakes, C. J. Fox, and B. A. Nejmeh. *Software Engineering in the Unix/C Environment*. Prentice Hall, Inc, 1991.
132. C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.
133. S. Frederick, G. Loewenstein, and T. O’Donoghue. Time discounting: A critical review. *Journal of Economic Literature*, 40(2):351–401, 2002.
134. M. Fredericks. Using defect tracking and analysis to improve software quality. Thesis (m.s.), Department of Computer Science, University of Maryland, 1999.
135. D. P. Freedman and G. M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews*. Dorset House Publishing, 1990.
136. W.-T. Fu and W. D. Gray. Memory versus perceptual-motor trade-offs in a blocks world task. In *Proceedings of the Twenty-second Annual Conference of the Cognitive Science Society*, pages 154–159, Hillsdale, NJ, 2000. Erlbaum.
137. H. Gall, M. Jazayeri, R. R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. In *Proceedings of the International Conference on Software Maintenance (ICSM’97)*, pages 160–166, 1997.
138. H. Gardner. *Intelligence Reframed*. Basic Books, 1999.
139. S. E. Gathercole, S. J. Pickering, C. Knight, and Z. Stegmann. Working memory skills and education attainment: Evidence from national curriculum assessments at 7 and 14 years of age. *Applied Cognitive Psychology*, 18(1):1–16, 2004.
140. S. A. Gelman and E. M. Markman. Categories and induction in young children. *Cognition*, 23:183–209, 1986.
141. R. A. Ghosh, R. Glott, B. Krieger, and G. Robles. Free/Libre and open source software survey and study, part 4: Survey of developers. Technical Report Deliverable D18: Final Report, University of Maastricht, June 2002.
142. A. Gierlinger, R. Forsyth, and E. Ofner. GEPARD: A parameterisable DSP core for ASICs. In *Proceedings ICSPAT’97*, pages 203–207, 1997.
143. G. Gigerenzer, P. M. Todd, and The ABC Research Group. *Simple Heuristics That Make Us Smart*. Oxford University Press, 1999.
144. D. T. Gilbert, R. W. Tafarodi, and P. S. Malone. You can’t not believe everything you read. *Journal of Personality and Social Psychology*, 65(2):221–233, 1993.
145. R. L. Glass. Persistent software errors. *IEEE Transactions on Software Engineering*, 7(2):162–168, Mar. 1981.
146. A. M. Glenberg and W. Epstein. Inexpert calibration of comprehension. *Memory & Cognition*, 15(1):84–93, 1987.
147. M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance (ICSM’00)*, pages 131–142, Oct. 2000.
148. R. A. Goldberg, S. Schwartz, and M. Stewart. Individual differences in cognitive processes. *Journal of Educational Psychology*, 69(1):9–14, 1977.
149. E. E. Grant and H. Sackman. An exploratory investigation of programmer performance under on-line and off-line conditions. *IEEE Transactions on Human Factors in Electronics*, 8(1):33–48, Mar. 1967.
150. T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.
151. P. Grice. *Studies in the Way of Words*. Harvard University Press, 1989.
152. R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL 4 Programming Language*. Prentice Hall, Inc, second edition, 1968.
153. A. J. M. Groenewegen and W. A. Wagenaar. Diagnosis in everyday situations: Limitations of performance at the knowledge level. Unit of Experimental Psychology, Leiden University, 1988.
154. D. Grune, H. E. Bel, C. J. H. Jacobs, and K. G. Langerdoen. *Modern Compiler Design*. John Wiley & Sons, Ltd, 2000.
155. R. Gupta, E. Mehofer, and Y. Zhang. Profile guided compiler optimizations. In Y. N. Srikant and P. Shankar, editors, *The Compiler Design Handbook: Optimizations and Machine Code Generation*, chapter 4, pages 143–174. CRC Press, 2002.
156. K. Hajek. Detection of logical coupling based on product release history. Thesis (m.s.), Technical University of Vienna, 1998.
157. G. S. Halford, W. H. Wilson, and S. Phillips. Processing capacity defined by relational complexity: Implications for comparative, developmental, and cognitive psychology. *Behavioral & Brain Sciences*, 21(6):803–831, 1998.
158. D. Z. Hambrick and R. W. Engle. Effect of domain knowledge, working memory capacity, and age on cognitive performance: An investigation of the knowledge-is-power hypothesis. *Cognitive Psychology*, 44(4):339–387, 2002.
159. K. R. Hammond, R. M. Hamm, J. Grassia, and T. Pearson. Direct comparison of the efficacy of intuitive and analytical cognition in expert judgment. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-17:753–770, Sept. 1987.
160. A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. Technical Report Research Report 96/3, Compaq Western Research Laboratory, 1996.
161. L. Hatton. *Safer C : Developing Software for High-integrity and Safety-critical Systems*. McGraw–Hill, 1995.
162. L. Hatton. The T-experiments: Errors in scientific software. *IEEE Computational Science & Engineering*, 4(2):27–38, Jan. 1997.
163. J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc, 1996.
164. M. Henricson and E. Nyquist. *Industrial Strength C++, Rules and Recommendations*. Prentice Hall, Inc, 1997.

165. R. N. A. Henson. *Short-term Memory for Serial Order*. PhD thesis, University of Cambridge, Nov. 1996.
166. D. M. B. Herbert and J. S. Burt. What do students remember? Episodic memory and the development of schematization. *Applied Cognitive Psychology*, 18(1):77–88, 2004.
167. D. S. Herrmann. *Software Safety and Reliability*. IEEE Computer Society, 1999.
168. R. Hertwig and G. Gigerenzer. The ‘conjunction fallacy’ revisited: How intelligent inferences look like reasoning errors. *Journal of Behavioral and Decision Making*, 12(2):275–305, 1999.
169. R. Hertwig and P. M. Todd. More is not always better: The benefits of cognitive limits. In L. Macchi and D. Hardman, editors, *The psychology of reasoning and decision making: A handbook*. John Wiley & Sons, Inc, 2000?
170. R. M. Hogarth and H. J. Einhorn. Order effects in belief updating: The belief-adjustment model. *Cognitive Psychology*, 24:1–55, 1992.
171. R. M. Hogarth, C. R. M. McKenzie, B. J. Gibbs, and M. A. Marquis. Learning from feedback: Exactness and incentives. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 17(4):734–752, 1991.
172. J. H. Holland, K. J. Holyoak, R. E. Nisbett, and P. R. Thagard. *Induction*. The MIT Press, 1989.
173. A. A. Hook, B. Brykczynski, C. W. McDonald, S. H. Nash, and C. Youngblut. A survey of computer programming languages currently used in the department of defense. Technical Report P-3054, Institute for Defense Analyse, Jan. 1995.
174. R. Hoosain. Correlation between pronunciation speed and digit span size. *Perception and Motor Skills*, 55:1128–1128, 1982.
175. R. Hoosain and F. Salili. Language differences, working memory, and mathematical ability. In M. M. Grunberg, P. E. Morris, and R. N. Sykes, editors, *Practical aspects of memory: Current research and issues*, volume 2, pages 512–517. John Wiley & Sons, Inc, 1988.
176. M. R. Horton. *Portable C Software*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.
177. C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 38–8, 2003.
178. C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic frequency and voltage scheduling. Technical Report DCS-TR-419, Department of Computer Science, Rutgers University, 2000.
179. Intel. *Desktop Performance and Optimization for Intel Pentium 4 Processor*. Intel, Inc, Feb. 2001.
180. ISO. *ISO 6160-1979(E) —Programming languages —PL/I*. ISO, 1979.
181. ISO. *ISO 1989-1985(E) —Programming languages —COBOL*. ISO, 1985.
182. ISO. *Implementation of ISO/IEC TR 10034:1990 Guidelines for the preparation of conformity clauses in programming language standards*. ISO, 1990.
183. ISO. *ISO/IEC 9945-1:1990 Information technology —Portable Operating System Interface (POSIX)*. ISO, 1990.
184. ISO. *ISO/IEC Guide 25:1990 General requirements for the competence of calibration and testing laboratories*. ISO, 1990.
185. ISO. *Implementation of ISO/TR 9547:1988 Programming language processors —Test methods —Guidelines for their development and acceptability*. ISO, 1991.
186. ISO. *ISO/IEC 10206:1991 Information technology —Programming languages —Extended Pascal*. ISO, 1991.
187. ISO. *ISO/IEC 1539:1992 Information technology —Programming languages —FORTRAN*. ISO, 1991.
188. ISO. *ISO/IEC 9075:1992(E) Information technology —Database languages —SQL*. ISO, 1992.
189. ISO. *ISO/IEC 8652:1995(E) Information technology —Programming languages —Annotated Ada Reference Manual*. ISO, 1995.
190. ISO. *ISO/IEC 10514-1:1996 Information technology —Programming languages —Part 1. Modula-2, Base language*. ISO, 1996.
191. ISO. *Implementation of ISO/IEC TR 10176:1997 Information technology —Guidelines for the preparation of programming language standards*. ISO, 1997.
192. ISO. *ISO/IEC 13816:1997 Information technology —Programming languages, their environments and system software interfaces —Programming language ISLISP*. ISO, 1997.
193. ISO. *ISO/IEC 13210:1999 Information technology —Requirements and guidelines for test methods specifications and test method implementation for measuring conformance to POSIX standards*. ISO, 1999.
194. ISO. *ISO/IEC 13751.2:2000 Information technology —Programming languages, their environments and system software interfaces —Programming language Extended APL*. ISO, 2000.
195. ISO. *ISO/IEC TR 15942:2000 Programming languages —Guide for the Use of the Ada Programming Language in High Integrity Systems*. ISO, 2000.
196. ISO. *ISO/IEC 9496:2003 CHILL —The ITU-T programming language*. ISO, 2003.
197. S. A. J. The seer-sucker theory: The value of experts in forecasting. *Technology Review*, pages 16–24, June-July 1980.
198. R. Jaeschke. *Portability and the C Language*. Hayden Books, 4300 West 62nd Street, Indianapolis, IN 46268, USA, 1989.
199. P. J. Jalics. COBOL on a PC: A new perspective on a language and its performance. *Communications of the ACM*, 30(2):142–154, Feb. 1987.
200. M. K. Johansen and T. J. Palmeri. Are there representational shifts during category learning? *Cognitive Psychology*, 45(4):482–553, 2002.
201. S. C. Johnson. A tour through the portable C compiler. In B. W. Kernighan and M. D. McIlroy, editors, *Unix Programmer’s Manual, 7th edition, Volume 2B*, chapter 33. Bell Laboratories, Murray Hill, NJ, Jan. 1979. Republished by Holt, Rinehart and Winston, New York, ISBN 0-03-061743-X, 1983.
202. C. Jones. *Programming Productivity*. McGraw-Hill Book Company, 1986.
203. D. M. Jones. The Model C Implementation. Knowledge Software Ltd, 1992.
204. D. M. Jones. The 7±2 urban legend. MISRA C 2002 conference <http://www.knosof.co.uk/cbook/misart.pdf>, Oct. 2002.
205. J. Jonides and C. M. Jones. Direct coding for frequency of occurrence. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 18(2):368–378, 1992.
206. M. Jørgensen. A review of studies on expert estimation of software development effort. *Journal of Systems and Software*, 70(1-2):37–60, 2004.
207. M. Jørgensen and D. I. K. Sjøberg. Impact of experience on maintenance skills. *Journal of Software Maintenance: Research and Practice*, 14(2):123–146, 2002.

208. N. P. Jouppi and P. Ranganathan. The relative importance of memory latency, bandwidth, and branch limits to performance. In *Proceedings of Workshop of Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997.
209. D. M. Kagan and J. M. Douthat. Personality and learning FORTRAN. *International Journal of Man-Machine Studies*, 22(4):395–402, 1985.
210. D. Kahneman, P. Slovic, and A. Tversky, editors. *Judgment under uncertainty: Heuristics and biases*. Cambridge University Press, 1982.
211. D. Kahneman and A. Tversky. On the psychology of prediction. In D. Kahneman, P. Slovic, and A. Tversky, editors, *Judgment under uncertainty: Heuristics and biases*, chapter 4, pages 48–68. Cambridge University Press, 1982.
212. D. Kahneman and A. Tversky. Subjective probability: A judgment of representativeness. In D. Kahneman, P. Slovic, and A. Tversky, editors, *Judgment under uncertainty: Heuristics and biases*, chapter 3, pages 32–47. Cambridge University Press, 1982.
213. D. Kahneman and A. Tversky. Choices, values, and frames. In D. Kahneman and A. Tversky, editors, *Choices, Values, and Frames*, chapter 1, pages 1–16. Cambridge University Press, 1999.
214. D. Kahneman and A. Tversky. Prospect theory: An analysis of decision under risk. In D. Kahneman and A. Tversky, editors, *Choices, Values, and Frames*, chapter 2, pages 17–43. Cambridge University Press, 1999.
215. S. Kahrs. Mistakes and ambiguities in the definition of standard ML. Technical Report LFCS report ECS-LFCS-93-257, University of Edinburgh, Scotland, Apr. 1993.
216. Y. Kareev. Seven (indeed, plus or minus two) and the detection of correlations. *Psychological Review*, 107(2):397–402, 2000.
217. R. Kelsey, W. Clinger, J. Rees, H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. A. IV, D. P. Friedman, E. Kohlbecker, G. L. Steele JR., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised⁵ report on the algorithmic language Scheme. Technical report, Feb. 1998.
218. C. K. Kemmerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4):493–503, 1999.
219. B. W. Kernighan and R. Pike. *The Practice of Programming*. Addison–Wesley, 1999.
220. D. E. Kieras and D. E. Meyer. An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. Technical Report TR-95/ONR-EPIC-5, University of Michigan, 1995.
221. T. Kistler and M. Franz. Continuous program optimization: A case study. Technical Report Technical Report No. 00-19, Department of Information and Computer Science, University of California, Irvine, 2000.
222. D. Klahr, W. G. Chase, and E. A. Lovelace. Structure and process in alphabetic retrieval. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 9(3):462–477, 1983.
223. J. Klayman and Y.-W. Ha. Confirmation, disconfirmation, and information in hypothesis testing. *Psychological Review*, 94(2):211–228, 1987.
224. J. Klayman, J. B. Soll, C. Gonz/alez-Vallejo, and S. Barlas. Overconfidence: It depends on how, what, and whom you ask. *Organizational Behavior and Human Decision Processes*, 79(3):216–247, 1999.
225. G. Klein. *Sources of Power*. The MIT Press, 1999.
226. J. L. Knetsch. The endowment effect and evidence of nonreversible indifference curves. In D. Kahneman and A. Tversky, editors, *Choices, Values, and Frames*, chapter 9, pages 171–179. Cambridge University Press, 1999.
227. D. E. Knuth. An empirical study of FORTRAN programs. *Software–Practice and Experience*, 1:105–133, 1971.
228. D. E. Knuth. The errors of TeX. *Software–Practice and Experience*, 19(7):607–685, 1989.
229. J. J. Koehler. The base rate fallacy reconsidered: Descriptive, normative and methodological challenges. *Behavior & Brain Sciences*, 19(1):1–17, 1996.
230. A. Koenig. *C Traps and Pitfalls*. Addison–Wesley, 1989.
231. A. Koriat. How do we know that we know? The accessibility model of the feeling of knowing. *Psychological Review*, 100(4):609–639, 1993.
232. A. Koriat, M. Goldsmith, and A. Pansky. Toward a psychology of memory accuracy. *Annual Review of Psychology*, 51:481–537, 2000.
233. S. M. Kosslyn and S. P. Shwartz. Empirical constraints on theories of visual imagery. In J. Long and A. D. Baddeley, editors, *Attention and Performance IX*, pages 241–260. Lawrence Erlbaum Associates, 1981.
234. R. J. Koubek, W. K. LeBold, and G. Salvendy. Predicting performance in computer programming courses. *Behavior and Information Technology*, 4(2):113–129, 1985.
235. C. B. Kreitzberg and B. Shneiderman. *The elements of FORTRAN style: techniques for effective programming*. Harcourt, Brace, Jovanovich, San Diego, CA, USA, 1972.
236. I. Krsul. Authorship analysis: Identifying the author of a program. Technical Report Purdue Technical Report CSD-TR-94-030, Department of Computer Sciences, Purdue University, 1994.
237. I. Krsul and E. H. Spafford. Authorship analysis: Identifying the author of a program. Technical Report Technical Report TR-96-052, Department of Computer Sciences, Purdue University, 1996.
238. L. E. Krueger. A theory of perceptual matching. *Psychological Review*, 85(4):278–304, 1978.
239. O. Laitenberger and J.-M. DeBaud. Perspective-based reading of code documents at Robert Bosch GmbH. Technical Report Technical Report ISERN-97-14, Fraunhofer Institute for Experimental Software Engineering, 1997.
240. O. Laitenberger and J.-M. DeBaud. An encompassing life-cycle centric survey of software inspection. Technical Report Technical Report ISERN-98-32, Fraunhofer Institute for Experimental Software Engineering, 1998.
241. O. Laitenberger, K. E. Emam, and T. Harbich. An internally replicated quasi-experimental comparison of checklist and perspective-based reading of code documents. Technical Report Technical Report ISERN-99-01, Fraunhofer Institute for Experimental Software Engineering, 1999.
242. J. Lakos. *Large Scale C++ Software Design*. Addison–Wesley, 1996.
243. T. K. Landauer. How much do people remember? Some estimates of the quantity of learned information in long-term memory. *Cognitive Science*, 10:477–493, 1986.
244. G. Langdale. *The Effect of Profile Choice and Profile Gathering Methods on Profile-Driven Optimization Systems*. PhD thesis, Carnegie Mellon University, Oct. 2003.

245. E. J. Langer. The illusion of control. *Journal of Personality and Social Psychology*, 32(2):311–328, 1975.
246. D. Lanneer, J. V. Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHES: Retargetable code generation for embedded DSP processors. In P. Merwedel and G. Goossens, editors, *Code generation for embedded processors*, chapter 5, pages 85–102. Kluwer Academic Publishers, July 1995.
247. K. A. Latorella. Investigating interruptions: Implications for flight-deck performance. Technical Report NASA/TM-1999-209707, NASA, Oct. 1999.
248. A. R. Lebeck. Cache conscious programming in undergraduate computer science. In D. Joyce, editor, *Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, volume 31.1 of *SIGCSE Bulletin*, pages 247–251, N. Y., Mar. 24–28 1999. ACM Press.
249. C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '97)*, pages 330–335. IEEE, Dec. 1997.
250. D. C. Lee, P. J. Crowley, J.-L. Baer, T. E. Anderson, and B. N. Bershad. Execution characteristics of desktop applications on Windows NT. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *ACM Computer Architecture News*, pages 27–38, New York, June 27–July 1 1998. ACM Press.
251. C. Leen, J. K. Lee, T. T. Hwang, and S.-C. Tsai. Compiler optimization on instruction scheduling for low power. In *Proceedings of the 13th International Symposium on System Synthesis (ISSS'00)*, 2000.
252. D. R. Lehman, R. O. Lempert, and R. E. Nisbett. The effects of graduate training on reasoning. *American Psychologist*, 43(6):431–442, 1988.
253. M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Wurski. Metrics and laws of software evolution - the nineties view. In *4th International Software Metrics Symposium (METRICS '97)*, pages 20–32. IEEE, Nov. 1997.
254. P. Lemaire, H. Abdi, and M. Fayol. The role of working memory resources in simple cognitive arithmetic. *European Journal of Cognitive Psychology*, 8(1):73–103, 1996.
255. T. C. Lethbridge. What knowledge is important to a software professional? *IEEE Computer*, 33(5):44–50, May 2000.
256. T. C. Lethbridge, S. E. Sim, and J. Singer. Software anthropology: Performing field studies in software companies. 2000.
257. S. Letovsky. Cognitive processes in program comprehension. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers*, pages 58–79. Ablex Publishing Corporation, 1986.
258. S. Letovsky. Cognitive processes in program comprehension. *The Journal of Systems and Software*, 7(4):325–339, Dec. 1987.
259. R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1):1–36, Jan. 1998.
260. R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1):1–36, Jan. 1998.
261. B. W. Leverett, R. G. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schatz, and W. A. Wulf. An overview of the production-quality compiler-compiler project. *Computer*, 13(8):38–49, 1980.
262. P. Lewicki, T. Hill, and E. Bizot. Acquisition of procedural knowledge about a pattern stimuli that cannot be articulated. *Cognitive Psychology*, 20:24–37, 1988.
263. E. Y. Li, H.-G. Chen, and W. Cheung. Total quality management in software development process. *The Journal of the Quality Assurance Institute*, 14(1):4–6 & 35–41, Jan. 2000.
264. S. Lichtenstein and B. Fishhoff. Do those who know more also know more about how much they know? *Organizational Behavior and Human Performance*, 20:159–183, 1977.
265. D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers*, pages 80–98. Ablex Publishing Corporation, 1986.
266. G. P. Logan. Toward an instance theory of automatization. *Psychological Review*, 95(4):492–527, 1988.
267. A. Loginov, S. H. Yong, S. Horowitz, and T. Reps. Debugging via run-time type checking. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering 4th International Conference (FASE 2001)*, pages 217–232. Springer-Verlag, Apr. 2001.
268. J. A. Lucy. *Language diversity and thought: A reformulation of the linguistic relativity hypothesis*. Cambridge University Press, 1992.
269. P. Lukowicz, E. A. Heinz, L. Prechelt, and W. F. Tichy. Experimental evaluation in computer science: a quantitative study. Technical Report iratr-1994-17, Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation, Feb. 1994.
270. A. R. Luria. *The mind of a mnemonist*. Harvard University Press, 1986.
271. J. N. MacGregor. Short-term memory capacity: Limitation or optimization? *Psychological Review*, 94(1):107–108, 1987.
272. L. MacKellar. Variations in productivity over the life span: A review and some implications. Technical Report IR-02-061, International Institute for Applied Systems Analysis, Austria, Sept. 2002.
273. C. M. MacLeod, E. B. Hunt, and N. N. Matthews. Individual differences in the verification of sentence-picture relationships. *Journal of Verbal Learning and Verbal Behavior*, 17:493–507, 1978.
274. W. T. Maddox and C. J. Bohil. Costs and benefits in perceptual categorization. *Memory & Cognition*, 28:597–615, 2000.
275. D. J. Magenheimer, L. Peters, K. W. Pettis, and D. Zuras. Integer multiplication and division on the HP precision architecture. *IEEE Transactions on Computers*, 37(8):980–990, 1988.
276. E. A. Maguire, D. G. Gadian, I. S. Johnsru, C. D. Good, J. Ashburner, R. S. J. Frackowiak, and C. D. Frith. Navigation-related structural change in the hippocampi of taxi drivers. *Proceedings of the National Academy of Sciences*, 97(8):4398–4403, 2000.
277. A. B. Markman and E. J. Wisniewski. Similar and different: The differentiation of basic-level categories. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 23(1):54–70, 1997.
278. M. Martin. Memory span as a measure of individual differences in memory capacity. *Memory & Cognition*, 6(2):194–198, 1978.
279. H. Massalin. Superoptimizer – A look at the smallest program. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 122–126. ACM Press, Oct. 1987.
280. E. Matias, I. S. MacKenzie, and W. Buxton. One-handed touch-typing on a QWERTY keyboard. *Human-Computer Interaction*, 11:1–27, 1996.

281. A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *ASPLOS-VI: Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, Oct. 1994.
282. D. C. McClelland. Testing for competence rather than for "intelligence". *American Psychologist*, 28:1–14, Jan. 1973.
283. M. McCloskey, A. Washburn, and L. Felch. Intuitive physics: The straight-down belief and its origin. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 9(4):636–649, 1983.
284. S. McConnell. *Code Complete*. Microsoft Press, 1993.
285. D. McFadden. Rationality for economists? *Journal of Risk and Uncertainty*, 19:73–105, 1999.
286. D. C. McFarlane. Interruption of people in human-computer interaction: A general unifying definition of human interruption and taxonomy. Technical Report NRL/FR/5510-97-9870, Naval Research Laboratory, Dec. 1997.
287. K. B. McKeithen, J. S. Reitman, H. H. Ruster, and S. C. Hirtle. Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13:307–325, 1981.
288. L. McMahan and R. Lee. Pathlengths of SPEC benchmarks for PA-RISC, MIPS, and SPARC. In *Proceedings of IEEE Comcon*, pages 481–490, Feb. 1993.
289. J. McMillan. Enhancing college student's critical thinking: A review of studies. *Research in Higher Education*, 26:3–29, 1987.
290. R. E. Melchers and M. V. Harrington. Human error in simple design tasks. Technical Report Civil Engineering Research Reports Report Number 31, Monash University, 1982.
291. R. C. Merkle. Energy limits to the computational power of the human brain. *Foresight Update*, 6, Aug. 1989.
292. S. Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley professional computing series. Addison-Wesley, Reading, MA, USA, 1992.
293. S. Meyers, C. K. Duby, and S. P. Reiss. Constraining the structure and style of object-oriented programs. Technical Report CS-93-12, Department of Computer Science, Brown University, Box 1910, Providence, Rhode Island 02912, U.S.A., Apr. 1993.
294. S. Meyers and M. Lejter. Automatic detection of C++ programming errors: Initial thoughts on a lint++. Technical Report Technical Report CS-91-51, Brown University, 1991.
295. S. Milgram. *Obedience to Authority*. McGraw-Hill, 1974.
296. G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63(2):81–97, 1956.
297. J. Miller, J. Daly, M. Wood, M. Roper, and A. Brooks. Statistical power and its subcomponents – missing and misunderstood concepts in empirical software engineering research. Technical Report EFOCS-15-94, Department of Computer Science, University of Strathclyde, Livingstone Tower, Richmond Street, Glasgow G1 1XH, UK, 1994.
298. J. Miller, M. Wood, M. Roper, and A. Brooks. Further experiences with scenarios and checklists. Technical Report EFOCS-20-94, University of Strathclyde, 1994.
299. MISRA. *Guidelines for the Use of the C Language in Vehicle Based Software*. Motor Industry Research Association, Nuneaton CV10 0TU, UK, 1998.
300. MISRA. *MISRA-C:2004 Guidelines for the Use of the C Language in Vehicle Based Software*. Motor Industry Research Association, Nuneaton CV10 0TU, UK, 2004.
301. MISRA. *MISRA-C++:2008 Guidelines for the use of the C++ language in critical systems*. Motor Industry Research Association, Nuneaton CV10 0TU, UK, 2008.
302. S. Mithen. *The Prehistory of the Mind*. Thames and Hudson, 1996.
303. A. Miyake and P. Shah. *Models of Working Memory: Mechanisms of Active Maintenance and Executive Control*. Cambridge University Press, 1999.
304. A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. Technical Report ALR-2002-003, Avaya Labs Research, Jan. 2002.
305. A. Mockus and D. M. Weiss. Predicting risk in software changes. *Bell Labs Technical Journal*, Apr.-June 2000.
306. T. Moher and G. M. Schneider. Methods for improving controlled experimentation in software engineering. In *Proceedings of the 5th international conference on Software engineering*, pages 224–233. IEEE Computer Society, Mar. 1981.
307. P. Monaghan. A corpus-based analysis of individual differences in proof-style. Thesis (m.s.), Centre for Cognitive Science, University of Edinburgh, 1995.
308. P. Monaghan. *Representation and Strategy in Reasoning: An Individual Differences Approach*. PhD thesis, University of Edinburgh, 2000.
309. S. Monsell. Task switching. *TRENDS in Cognitive Science*, 7(3):134–140, 2003.
310. J. E. Moore and L. A. Burke. How to turn around 'turnover culture' in IT. *Communications of the ACM*, 45(2):73–78, 2002.
311. T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the USENIX 2nd Symposium on Operating Systems Designed and Implementation*, pages 3–17. USENIX Association, Oct. 1996.
312. S. S. Muchnick. *Advances Compiler Design & Implementation*. Morgan Kaufmann Publishers, 1997.
313. G. L. Murphy and D. L. Medin. The role of theories in conceptual coherence. *Psychological Review*, 92(3):289–315, 1985.
314. I. B. Myers, M. H. McCaulley, N. L. Quenk, and A. L. Hammer. *A Guide to the Development and Use of the Myers-Briggs Type Indicator*. Consulting Psychologists Press, third edition, 1998.
315. C. R. Mynatt, M. E. Doherty, and W. Dragan. Information relevance, working memory, and the consideration of alternatives. *Quarterly Journal of Experimental Psychology*, 46A(4):759–778, 1993.
316. C. R. Mynatt, M. E. Doherty, and R. D. Tweney. Confirmation bias in a simulated research environment. *Quarterly Journal of Experimental Psychology*, 29:85–95, 1997.
317. R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 40–51, Dec. 2001.
318. NASA. NASA GB-1740.13-96: NASA guidebook for safety critical software - analysis and development. Technical report, NASA Glenn Research Center, 1996.
319. G. C. Necula, S. McPeak, and W. Weimer. Taming C pointers. www.cs.berkeley.edu/~necula, 2004.

320. K. M. Nelson, H. J. Nelson, and M. Ghods. Understanding the personal competencies of IS support experts: Moving towards the E-business future. In *34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 8*. IEEE, Jan. 2001.
321. A. Newell and P. S. Rosenbloom. Mechanisms of skill acquisition and the power law of practice. In J. R. Anderson, editor, *Cognitive skills and their acquisition*, pages 1–54. Erlbaum, Hillsdale, NJ, 1981.
322. D. M. Nichols and M. B. Twidale. Usability and open source software. Technical Report Working Paper 10/02, University of Waikato, 2002.
323. R. S. Nickerson, D. N. Perkins, and E. E. Smith. *The Teaching of Thinking*. Erlbaum, Hillsdale NJ, 1985.
324. B. K. Nirmal. *PROGRAMMING STANDARDS and GUIDELINES: COBOL edition*. Prentice-Hall, Inc, 1987.
325. R. E. Nisbett, D. H. Krantz, C. Jepson, and Z. Kunda. The use of statistical heuristics in everyday inductive reasoning. *Psychological Review*, 90(4):339–363, 1983.
326. R. E. Nisbett and A. Norenzayan. Culture and cognition. In D. Medin and H. Pashler, editors, *Stevens' Handbook of Experimental Psychology, Volume Two: Memory and Cognitive Processes*, chapter 13. John Wiley & Sons, third edition, Apr. 2002.
327. R. M. Nosofsky. Exemplar-based accounts of relations between classification, recognition and typicality. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 14(4):700–708, 1988.
328. K. Oberauer, H.-M. Süß, O. Wilhelm, and W. W. Wittmann. The multiple faces of working memory: Storage, processing, supervision, and coordination. *Intelligence*, 31:167–193, 2003.
329. J. Oberlander, R. Cox, P. Monaghan, K. Stenning, and R. Tobin. Individual differences in proof structures following multimodal logic teaching. In *Proceedings of the 18th Annual Meeting of the Cognitive Science Society*, pages 201–206, 1996.
330. M. C. Ohlsson. Utilisation of historical data for controlling and improving software development. Licentiate, Lund Institute of Technology, Sweden, 1999.
331. M. C. Ohlsson. *Controlling Fault-Prone Components for Software Evolution*. PhD thesis, Lund Institute of Technology, Sweden, 2001.
332. D. N. Osherson, O. Wilkie, E. Shafir, E. E. Smith, and A. López. Category-based induction. *Psychological review*, 97(2):185–200, 1990.
333. P. W. Paese and J. A. Sniezek. Influences on the appropriateness of confidence in judgment: Practice, effort, information, and decision-making. *Organizational Behavior and Human Decision Processes*, 48:100–130, 1991.
334. S. E. Palmer. *Vision Science: Photons to Phenomenology*. The MIT Press, 1999.
335. S. Paoli. C++ coding standard specification. Technical Report CERN-UCO/1999/207, CERN, Jan. 2000.
336. R. E. Park. Software size measurement: A framework for counting source statements. Technical Report CMU/SEI-92-TR-20, Software Engineering Institute, Sept. 1992.
337. H. E. Pashler. *The Psychology of Attention*. The MIT Press, 1999.
338. M. A. Paskin. Maximum entropy probabilistic logic. Technical report, University of California, Berkeley, USA, 2002.
339. A. Patel. Auditors' belief revision: Recency effects of contrary and supporting audit evidence and source reliability. *The Auditors Report*, 24(3), 2001.
340. J. W. Payne, J. R. Bettman, and E. J. Bettman. *The Adaptive Decision Maker*. Cambridge University Press, 1993.
341. M. J. Pazzani. Influence of prior knowledge on concept acquisition: Experimental and computational results. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 17(3):416–432, 1991.
342. N. Pennington. Comprehension strategies in programming. In G. Olson, S. Shepard, and E. Soloway, editors, *Empirical Studies of programmers: Second Workshop*, chapter 7, pages 100–113. Ablex Publishing, 1987.
343. N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.
344. D. E. Perry, N. A. Staudenmayer, and L. G. Votta Jr. People, organizations, and process improvement. *IEEE Software*, 11(4):36–45, July 1994.
345. D. E. Perry, N. A. Staudenmayer, and L. G. Votta Jr. Understanding and improving time usage in software development. In A. Fuggetta and A. L. Wolf, editors, *Trends in Software Process*, chapter 5. John Wiley & Sons, 1996.
346. D. E. Perry and C. S. Stieg. Software faults in evolving a large, real-time system: a case study. In *Proceedings of the 1993 European Software Engineering Conference*, pages 48–67, 1993.
347. S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International, 1987.
348. S. Pinker. *How the Mind Works*. Penguin, 1997.
349. R. Ploetzner and K. VanLehn. The acquisition of qualitative physics knowledge during textbook-based physics trainings. *Cognition and Instruction*, 15(2):169–205, 1997.
350. T. Plum. *Reliable data structures in C*. Plum Hall, 1985.
351. T. Plum. *C Programming guidelines*. Plum Hall, 1989.
352. T. Plum and D. Saks. *C++ Programming Guidelines*. Plum Hall, 1991.
353. K. R. Popper. *Conjectures and Refutations*. Routledge, 1969.
354. A. Porter, H. Siy, and L. Votta. A review of software inspections. In M. Zelkowitz, editor, *Advances in Computers 42*, pages 39–76. Academic Press, 1996.
355. A. Porter and L. Votta. Comparing detection methods for software requirements inspections: A replication using professional subjects. *Empirical Software Engineering*, 3(4):355–379, 1998.
356. A. A. Porter, H. Siy, A. Mockus, and L. G. Votta. Understanding the sources of variation in software inspections. Technical Report CS-TR-3762, University of Maryland, College Park, Jan. 1997.
357. A. A. Porter, H. P. Siy, C. A. Toman, and L. G. Votta. An experiment to assess the cost-benefits of code inspections in large scale software development. *IEEE Transactions on Software Engineering*, 23(6):329–346, June 1997.
358. POSC. *POSC Base Computer Standards: version 2*. Prentice Hall, Inc, 1994.
359. A. Postma, R. Izendoorn, and E. H. F. De Haan. Sex differences in object location memory. *Brain and Cognition*, 36:334–345, 1998.
360. E. M. Pothos and N. Chater. Rational categories. In *Proceedings of the Twentieth Annual Conference of the Cognitive Science Society*, pages 848–853, 1998.

361. L. Prechelt. Why we need an explicit forum for negative results. *Journal of Universal Computer Science*, 3(9):1074–1083, 1997.
362. L. Prechelt. The 28:1 Grant/Sackman legend is misleading, or: How large is interpersonal variation really? Technical Report iratr-1999-18, Universität Karlsruhe, 1999.
363. L. Prechelt. Comparing Java vs. C/C++ efficiency differences to interpersonal differences. *Communications of the ACM*, 42(10):109–112, Oct. 1999.
364. L. Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx and Tcl for a string processing program. Technical Report Technical Report 2000-5, Universität Karlsruhe, Fakultät für Informatik, 2000.
365. L. Prechelt, G. Malpohl, and M. Phippsen. JPlag: Finding plagiarisms among a set of programs. Technical Report Technical Report 2000-1, Universität Karlsruhe, Fakultät für Informatik, 2000.
366. C. C. Presson and D. R. Montello. Updating after rotational and translational body movements: coordinate structure of perspective space. *Perception*, 23:1447–1455, 1994.
367. T. A. Proebsting and B. G. Zorn. Programming shorthand. Technical Report Technical Report MSR-TR-2000-03, Microsoft Research, 2000.
368. J. B. Proffitt, J. D. Coley, and D. L. Medin. Expertise and category-based induction. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 26(4):811–828, 2000.
369. S. Qualline. *C Elements of Style*. M&T Books, 1992.
370. M. Rabin and J. Schrag. First impressions matter: A model of confirmation bias. *Quarterly Journal of Economics*, 114:37–82, 1999.
371. H. Rabinowitz and C. Schaap. *Portable C*. Prentice Hall, Inc, 1990.
372. D. Raffo, J. Settle, and W. Harrison. Investigating financial measures for planning software IV&V. Technical Report TR-99-05, Portland State University, 1999.
373. J. Ranade and A. Nash. *The Elements of C Programming Style*. McGraw-Hill, Inc, 1992.
374. P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.
375. J. Reason. *Human Error*. Cambridge University Press, 1990.
376. A. S. Reber and S. M. Kassir. On the relationship between implicit and explicit modes in the learning of a complex rule structure. *Journal of Experimental Psychology: Human Learning and Memory*, 6(5):492–502, 1980.
377. D. J. Reifer. Qualifying the debate: Ada vs C++. *Crosstalk: The Journal of Defense Software Engineering*, 1996.
378. M. Richards and C. Whitby-Stevens. *BCPL—the language and its compiler*. Cambridge University Press, 1979.
379. L. J. Rips. Inductive judgments about natural categories. *Journal of Verbal Learning and Verbal Behavior*, 14:665–681, 1975.
380. L. J. Rips, E. J. Shoben, and E. E. Smith. Semantic distance and the verification of semantic relations. *Journal of Verbal Learning and Verbal Behavior*, 12:1–20, 1973.
381. D. M. Ritchie. The development of the C language. *Second History of Programming Languages conference*, 1993.
382. A. D. Robison. Impact of economics on compiler optimization. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 1–10. ACM Press, 2001.
383. R. D. Rogers and S. Monsell. Costs of a predictable switch between simple cognitive tasks. *Journal of Experimental Psychology: General*, 124(2):207–231, 1995.
384. R. Ronen, A. Mendelson, K. Lai, S.-L. Lu, F. Pollack, and J. P. Shen. Coming challenges in microarchitecture and architecture. *Proceedings of the IEEE*, 89(3):325–340, 2001.
385. E. Rosch, C. B. Mervis, W. D. Gray, D. M. Johnson, and P. Boyes-Braem. Basic objects in natural categories. *Cognitive Psychology*, 8:382–439, 1976.
386. L. Ross, M. R. Lepper, and M. Hubbard. Perseverance in self-perception and social perception: Biased attributional processes in the debelieving paradigm. *Journal of Personality and Social Psychology*, 32(5):880–892, 1975.
387. D. C. Rubin and A. E. Wenzel. One hundred years of forgetting: A quantitative description of retention. *Psychological Review*, 103(4):734–760, 1996.
388. J. S. Rubinstein, D. E. Meyer, and J. E. Evans. Executive control of cognitive processes in task switching. *Journal of Experimental Psychology: Human Perception and Performance*, 27(4):763–797, 2001.
389. R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. Technical Report USC-CS-92-524, University of California, Berkeley, Sept. 1992.
390. R. Samuels, S. Stich, and L. Faucher. Reason and rationality. In I. Niiniluoto, M. Sintonen, and J. Wolenski, editors, *Handbook of Epistemology*, pages 131–179. Dordrecht:Kluwer, 2004.
391. W. Scacchi. Understanding software productivity. In D. Hurley, editor, *Advances in Software Engineering and Knowledge Engineering*, volume 4, pages 37–70. World Scientific, 1995.
392. D. A. Schkade and D. N. Kleinmuntz. Information displays and choice processes: Differential effects of organization, form, and sequence. *Organizational Behavior and Human Decision Processes*, 57:319–337, 1994.
393. W. Schneider. Training high-performance skills: Fallacies and guidelines. *Human Factors*, 27(3):285–300, 1985.
394. J. W. Schoonard and S. J. Boies. Short type: A behavior analysis of typing and text entry. *Human Factors*, 17(2):203–214, 1975.
395. C. D. Schunn, L. M. Reder, A. Nhouyvanisvong, D. R. Richards, and P. J. Strohffolino. To calculate or not to calculate: A source activation confusion model of problem familiarity’s role in strategy selection. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 23(1):3–29, 1997.
396. P. Sedlmeier, R. Hertwig, and G. Gigerenzer. Are judgments of the positional frequencies of letters systematically biased due to availability? *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 24(3):754–770, 1998.
397. T. M. Shaft and I. Vessey. The relevance of application domain knowledge: Characterizing the computer program comprehension process. *Journal of Management Information Systems*, 15(1):51–78, 1998.
398. U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, Aug. 2001.
399. B. A. Sheil. The psychological study of programming. *ACM Computing Surveys*, 13(1):101–120, Mar. 1981.
400. R. N. Shepard, C. I. Hovland, and H. M. Jenkins. Learning and memorization of classifications. *Psychological Monographs: General and Applied*, 75(15):1–39, 1961.

401. R. N. Shepard and J. Metzler. Mental rotation of three-dimensional objects. *Science*, 171:701–703, Feb. 1971.
402. T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report CS99-630, University of California, San Diego, Aug. 1999.
403. R. J. Shiller. *Irrational Exuberance*. Princeton University Press, 2000.
404. B. Shneiderman. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop Publishers, Inc, 1980.
405. S. M. Shugan. The cost of thinking. *Journal of Consumer Research*, 7:99–111, Sept. 1980.
406. A. Sides, D. Osherson, N. Bonini, and R. Viale. On the reality of the conjunction fallacy. *Memory & Cognition*, 30(2):191–198, 2002.
407. J. G. Siek, J. M. Squyres, and A. Lumsdaine. The laboratory for scientific computing (LSC): Coding standards. Technical report, University of Notre Dame, Apr. 2000.
408. S. Silberman. The geek syndrome. *Wired*, 9(12), Dec. 2001.
409. Silicon Graphics. *C Language Reference Manual*. Silicon Graphics, Inc, 007-0701-130 edition, 1999.
410. S. E. Sim, C. L. A. Clarke, and R. C. Holt. Archetypal source code searches: A survey of software developers and maintainers. In *Proceedings of the Sixth International Workshop on Program Comprehension*, pages 180–187, June 1998.
411. S. E. Sim and R. C. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *Proceedings of the Twentieth International Conference on Software Engineering*, pages 361–370, Apr. 1998.
412. H. A. Simon. *Models of Bounded Rationality: Behavioral Economics and Business Organization*. The MIT Press, 1982.
413. I. Simonson. Choice based on reasons: The case of attraction and compromise effects. *Journal of Consumer Research*, 16:158–173, Sept. 1989.
414. I. Simonson and A. Tversky. Choice in context: Tradeoff contrast and extremeness aversion. *Journal of Marketing Research*, 29:281–295, 1992.
415. D. I. K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, E. Karahasanovic, E. F. Koren, and M. Vokác. Conducting realistic experiments in software engineering. In *Proceedings of the 2002 International Symposium on Empirical Software Engineering (ISESE'02)*, pages 17–26, Oct. 2002.
416. D. I. K. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanović, N.-K. Liborg, and A. C. Rekdal. A survey of controlled experiments in software engineering. Technical Report 2004-4, SIMULA Research Laboratory, 2004.
417. V. Skirbekk. Age and individual productivity: A literature survey. Technical Report MPIDR Working Paper WP 2003-028, Max Planck Institute for Demographic Research, Aug. 2003.
418. N. J. Slamecka and P. Graf. The generation effect: Delineation of a phenomenon. *Journal of Experimental Psychology: Human Learning and Memory*, 4(6):592–604, 1978.
419. N. T. Slingerland and A. J. Smith. Measuring the performance of multimedia instruction sets. Technical Report UCB/CSD-00-1125, University of California Berkeley, USA, Dec. 2000.
420. M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on multiple issue. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 290–302, 1989.
421. S. Sonnentag. Excellent software professionals: experience, work activities, and perception by peers. *Behaviour & Information Technology*, 14(5):289–299, 1995.
422. F. Spadini, M. Fertig, and S. J. Patel. Characterization of repeating dynamic code fragments. Technical Report CRHC-02-09, University of Illinois at Urbana-Champaign, 2002.
423. A. Spector and I. Biederman. Mental set and shift revisited. *American Journal of Psychology*, 89:669–679, 1976.
424. D. Sperber and D. Wilson. *Relevance: Communication and Cognition*. Blackwell Publishers, second edition, 1995.
425. D. Spuler. *C++ and C debugging, testing and reliability*. Prentice Hall, Inc, 1994.
426. R. M. Stallman. *Using the GNU Compiler Collection*. Free Software Foundation, Mar. 2004.
427. D. Stamovlasis and G. Tsaparlis. Non-linear analysis of the effect of working-memory capacity on organic-synthesis problem solving. *Chemistry Education: Research and Practice in Europe*, 1(3):375–380, 2000.
428. L. Standing, J. Conezio, and R. N. Haber. Perception and memory for pictures: Single-trial learning of 2500 visual stimuli. *Psychonomic Science*, 19(2):73–74, 1970.
429. Standish Group. The chaos report. Technical report, The Standish Group, 1995.
430. S. Sternberg. Memory-scanning: Mental processes revealed by reaction-time experiments. *American Scientist*, 57(4):421–457, 1969.
431. A. Stevens and P. Coupe. Distortions in judged spatial relations. *Cognitive Psychology*, 10:422–437, 1978.
432. T. R. Stewart and C. M. Lusk. Seven components of judgmental forecasting skill: Implications for research and improving forecasts. *Journal of Forecasting*, 13:579–599, 1994.
433. D. Straker. *C-Style standards and guidelines*. Prentice Hall, Inc, 1992.
434. M. Strathern. 'Improving ratings': audit in the British university system. *European Review*, 5(3):305–321, 1997.
435. B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1999.
436. K. Sullivan, P. Chalasani, and S. Jha. Software design decisions as real options. Technical Report Technical Report 97-14, University of Virginia, June SULLIV 1.PDF.
437. H.-M. Süß, K. Oberauer, W. W. Wittmann, O. Wilhelm, and R. Schulze. Working memory capacity and intelligence: An integrative approach based on brunswick symmetry. Technical report, Universität Mannheim, 1996.
438. H. Sutter and A. Alexandrescu. *C++ Coding Standards: Rules, Guidelines, and Best Practices*. Addison Wesley, 2004.
439. E. B. Swanson. IS "maintainability": Should it reduce the maintenance effort. *The DATA BASE for Advances in Information Systems*, 30(1):65–76, 1999.
440. H. L. Swanson. What develops in working memory? A life span perspective. *Developmental Psychology*, 35(4):986–1000, 1999.
441. J. Sweller, J. F. Mawer, and M. R. Ward. Development of expertise in mathematical problem solving. *Journal of Experimental Psychology: General*, 112(4):639–661, 1983.
442. J. W. Tanaka and M. Taylor. Object categories and expertise: Is the basic level in the eye of the beholder. *Cognitive Psychology*, 23:457–482, 1991.

443. S. E. Taylor and J. D. Brown. Illusion and well-being: A social psychological perspective on mental health. *Psychological Bulletin*, 103(2):193–210, 1988.
444. B. E. Teasley, L. M. Leventhal, C. R. Mynatt, and D. S. Rohlman. Why software testing is sometimes ineffective: Two applied studies of positive test strategy. *Journal of Applied Psychology*, 79(1):142–155, 1994.
445. D. Tennenhouse. It's time to get physical. In *20th IEEE Real-Time Systems Symposium (RTSS'99)*, Dec. 1999.
446. K. Tentori, D. Osherson, L. Hasher, and C. May. Wisdom and ageing: Irrational preferences in college students but not older adults. *Cognition*, 81(3):B87–B99, 2001.
447. P. E. Tetlock. Accountability: The neglected social context of judgment and choice. *Research in Organizational Behavior*, 7:297–332, 1985.
448. P. E. Tetlock. An alternative metaphor in the study of judgment and choice: People as politicians. *Theory and Psychology*, 1(4):451–475, 1991.
449. P. E. Tetlock, O. V. Kristel, S. B. Elson, M. C. Green, and J. S. Lerner. The psychology of the unthinkable: Taboo trade-offs, forbidden base rates, and heretical counterfactuals. *Journal of Personality and Social Psychology*, 78:853–870, 2000.
450. Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*. Texas Instruments, spru189f edition, Oct. 2000.
451. Texas Instruments. *TMS320C6000 Programmer's Guide*. Texas Instruments, Inc, spru196d edition, Mar. 2000.
452. T. A. Thayer, M. Lipow, and E. C. Nelson. *Software Reliability*. North-Holland Publishing Company, 1978.
453. P. Thompson. Margaret Thatcher: a new illusion. *Perception*, 9:483–484, 1980.
454. M. Thorup. All structured programs have small tree-width and good register allocation. *Information and Computation*, 142(2):159–181, 1998.
455. H. Tomiyama and H. Yasuura. Optimal code placement of embedded software for instruction caches. In *European Design and Test Conference (ED&TC '96)*, pages 96–101. IEEE, Mar. 1996.
456. J. Torrellas, C. Xia, and R. L. Daigle. Optimizing the instruction cache performance of the operating system. *IEEE Transactions on Computers*, 47(12):1363–1381, 1998.
457. C.-W. Tseng. Software support for improving locality in advanced scientific codes. Technical Report CS-TR-4168, University of Maryland, 2000.
458. R. M. Tubbs, W. F. Messier Jr., and W. R. Knechel. Recency effects in the auditor's belief revision process. *The Accounting Review*, 65(2):452–460, Apr. 1990.
459. J. Turley. Embedded processors. www.extremetech.com, Jan. 2002.
460. R. T. Turley and J. M. Bieman. Competencies of exceptional and nonexceptional software engineers. *The Journal of Systems and Software*, 28(1):19–38, Jan. 1995.
461. A. Tversky. Elimination by aspects: A theory of choice. *Psychological Review*, 79(4):281–299, 1972.
462. A. Tversky and D. Kahneman. Availability: A heuristic for judging frequency and probability. In D. Kahneman, P. Slovic, and A. Tversky, editors, *Judgment under uncertainty: Heuristics and biases*, chapter 11, pages 163–178. Cambridge University Press, 1982.
463. A. Tversky and D. Kahneman. Judgment under uncertainty: Heuristics and biases. In D. Kahneman, P. Slovic, and A. Tversky, editors, *Judgment under uncertainty: Heuristics and biases*, chapter 1, pages 3–20. Cambridge University Press, 1982.
464. A. Tversky and D. Kahneman. Judgments of and by representativeness. In D. Kahneman, P. Slovic, and A. Tversky, editors, *Judgment under uncertainty: Heuristics and biases*, chapter 6, pages 84–98. Cambridge University Press, 1982.
465. A. Tversky, S. Sattath, and P. Slovic. Contingent weighting in judgment and choice. In D. Kahneman and A. Tversky, editors, *Choices, Values, and Frames*, chapter 28, pages 503–517. Cambridge University Press, 1999.
466. A. Tversky and I. Simonson. Context-dependent preferences. In D. Kahneman and A. Tversky, editors, *Choices, Values, and Frames*, chapter 29, pages 518–527. Cambridge University Press, 1999.
467. R. D. Tweney, M. E. Doherty, W. J. Worner, D. P. Pliske, C. R. Mynatt, K. A. Gross, and D. L. Arkkelin. Strategies of rule discovery in an inference task. *Quarterly Journal of Experimental Psychology*, 32:109–123, 1980.
468. U.S. DoD. Memorandum on the use of the Ada programming language. Technical report, U.S. Department of Defence, Apr. 1997.
469. V. Živojnović, J. M. Velarde, C. Schläger, and H. Meyr. DSP-STONE: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing and Technology (ICSPAT'94)*, 1994.
470. T. van Gelder. Penicillin for the mind? Reason, education and cognitive science. Technical Report Preprint No. 1/98, University of Melbourne Department of Philosophy, 1998.
471. T. van Gelder and A. Bulka. Reason!: Improving informal reasoning skills. In *Proceedings of the Australian Computers in Education Conference*, 2000.
472. C. Van Rooy, C. Stough, A. Pipingas, C. Hocking, and R. B. Silberstein. Spatial working memory and intelligence Biological correlates. *Intelligence*, 29:275–292, 2001.
473. A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. *Algol 68*. Springer-Verlag, 1976.
474. S. P. VanderWiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.
475. H. VanLehn. *Mind Bugs: The Origins of Procedural Misconceptions*. The MIT Press, 1990.
476. F. J. Varela, E. Thompson, and E. Rosch. *The Embodied Mind: Cognitive Science and Human Experience*. The MIT Press, 1999.
477. K. L. Verco and M. J. Wise. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In *Proceedings of First Australian Conference on Computer Science Education*, 1996.
478. P. Verghese and D. G. Pelli. The information capacity of visual attention. *Vision Research*, 32(5):983–995, 1992.
479. P. Verhaeghen and J. Cerella. Ageing, executive control, and attention: a review of meta-analyses. *Neuroscience and Biobehavioral Reviews*, 26(7):849–857, 2002.
480. P. Vickers. *CAITLIN: Implementation of a Musical Program Auralisation System to Study the Effects of Debugging Tasks as Performed by Novice Pascal Programmers*. PhD thesis, Loughborough University, 1999.
481. G. Visaggio. Value-based decision model for renewal processes in software maintenance. Technical Report ISERN-99-06, Department of Informatics, University of Bari, Italy, 1999.

482. J. Voas, L. Morell, and K. Miller. Using dynamic sensitivity analysis to assess testability. www.cigital.com, 1991.
483. A. von Mayrhauser and A. M. Vans. From program comprehension to tool requirements in an industrial environment. In *Proceedings Second Workshop on Program Comprehension*, pages 78–86, July 1993.
484. A. von Mayrhauser, A. M. Vans, and S. Lang. Program comprehension and enhancement of software. In *Proceedings IFIP World Computing Congress - Information Technology and Knowledge Engineering*, 1998.
485. L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly & Associates, Inc, 3rd edition, 2000.
486. S. R. Walli. The myth of application source-code conformance. *Standard View*, 4(2):94–99, June 1996.
487. P. C. Wason. On the failure to eliminate hypothesis in a conceptual task. *Quarterly Journal of Experimental Psychology*, XII:129–140, 1960.
488. Watcom. *Watcom C Language Reference*. Sybase, Inc, 11.0c edition, 2000.
489. T. White House. Guidelines and discount rates for benefit-cost analysis of federal programs. Technical Report OMB Circular A-94, US Government, 1992.
490. W. A. Wickelgren. Size of rehearsal group and short-term memory. *Journal of Experimental Psychology*, 68(4):413–419, 1964.
491. E. Wiles. Economic models of software reuse: A survey, comparison and partial validation. Technical Report UWA-DCS-99-032, Department of Computer Science, University of Wales, Aberystwyth, 1999.
492. J. Wiley. Expertise as mental set: The effects of domain knowledge in creative problem solving. *Memory & Cognition*, 26(4):716–730, 1998.
493. T. C. Willoughby. Are programmers paranoid? In *Proceedings of the Tenth Annual SIGCPR Conference*, pages 47–54, June 1972.
494. T. D. Wilson, S. Lindsey, and T. Y. Schooler. A model of dual attitudes. *Psychological Review*, 107(1):101–126, 2000.
495. J. C. Wise, D. L. Hannaman, P. Kozumplik, E. Franke, and B. L. Leaver. Methods to improve cultural communication skills in special operations forces. Technical Report ARI Contract Report 98-06, United States Army Research Institute for the Behavioral and Social Sciences, July 1998.
496. J. Withey. Investment analysis of software assets for product lines. Technical Report CMU/SEI-96-TR-010, Software Engineering Institute, Carnegie Mellon University, Nov. 1996.
497. WL | Delft Hydraulics. Programmer's guide C programming rules. Technical Report OMS report number 2001-02, WL | Delft Hydraulics, Nov. 2001.
498. M. Wolfe. How compilers and tools differ for embedded systems. In *Proceedings of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2005*, pages 1–4. ACM, Sept. 2005.
499. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, New York, June 22–24 1995. ACM Press.
500. M. Yang, G.-R. Uh, and D. B. Whalley. Efficient and effective branch reordering using profile data. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(6):667–697, 2002.
501. J. J. Yi and D. J. Lilja. Improving processor performance by simplifying and bypassing trivial computations. In *International Conference on Computer Design (ICCD'02)*, pages 462–467, Sept. 2002.
502. W. D. Yu. A software fault prevention approach in coding and root cause analysis. *Bell Labs Technical Journal*, Apr.-June 1998.
503. R. Yung. Evaluation of a commercial microprocessor. Technical Report SMLI TR-98-65, Sun Microsystems, 1998.
504. S. F. Zeigler. Comparing development costs of C and Ada. Technical report, Rational Software Corporation, Mar. 1995.
505. M. V. Zelkowitz and D. Wallace. Experimental models for validating computer technology. *IEEE Computer*, 31(5):23–31, May 1998.
506. J. Zobel. Reliable research: Towards experimental standards for computer science. In *Proceedings of the 21st Australian Computer Science Conference*, pages 217–229, Feb. 1998.