Technical Report
CMU/SEI-92-TR-020
ESC-TR-92-020

# Software Size Measurement: A Framework for Counting Source Statements

**Robert E. Park**

with the Size Subgroup of the Software Metrics Definition Working Group and the Software Process Measurement Project Team

**Technical Report**
CMU/SEI-92-TR-020
ESC-TR-92-020
September 1992

# Software Size Measurement: A Framework for Counting Source Statements

Robert E. Park
with the
Size Subgroup of the Software Metrics Definition Working Group
and the
Software Process Measurement Project Team

**Software Engineering Institute**
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

# Table of Contents

**List of Figures** v

**Preface** ix

**Acknowledgments** xi

**1. Introduction** 1
- 1.1. Scope 1
- 1.2. Objective and Audience 2
- 1.3. Relationship to Other Work 2
- 1.4. Endorsement of Specific Measures 3

**2. A Framework for Size Definition** 5
- 2.1. Definition Checklists 7
- 2.2. Supplemental Rules Forms 9
- 2.3. Recording and Reporting Forms 10

**3. A Checklist for Defining Source Statement Counts** 13
- 3.1. Describing Statement Counts 13
- 3.2. Modifying the Checklist to Meet Special Needs 21
- 3.3. Assigning Rules for Counting and Classifying Statement Types 21

**4. Size Attributes and Values: Issues to Consider When Creating Definitions** 23
- 4.1. Statement Type 23
  - 4.1.1. Executable statements 24
  - 4.1.2. Declarations 24
  - 4.1.3. Compiler directives 26
  - 4.1.4. Comments 27
  - 4.1.5. Blank lines 28
  - 4.1.6. Other statement types 28
  - 4.1.7. Order of precedence 28
  - 4.1.8. General guidance 29
- 4.2. How Produced 29
  - 4.2.1. Programmed 30
  - 4.2.2. Generated with source code generators 30
  - 4.2.3. Converted with automated translators 30
  - 4.2.5. Modified 32
  - 4.2.6. Removed 34
- 4.3. Origin 34
  - 4.3.1. New work: no prior existence 35
  - 4.3.2. Prior work 35
- 4.4. Usage 37
  - 4.4.1. In or as part of the primary product 38
  - 4.4.2. External to or in support of the primary product 38

4.5. Delivery 39
4.5.1. Delivered as source 39
4.5.2. Delivered in compiled or executable form, but not as source 40
4.5.3. Not delivered 41
4.6. Functionality 42
4.6.1. Operative statements 42
4.6.2. Inoperative statements (dead, bypassed, unused, unreferenced, or unaccessed) 42
4.7. Replications 45
4.7.1. Master source statements (originals) 45
4.7.2. Physical replicates of master statements, stored in the master code 46
4.7.3. Copies inserted, instantiated, or expanded when compiling or linking 46
4.7.4. Postproduction replicates 47
4.8. Development Status 48
4.9. Language 49
4.10. Clarifications (general) 50
4.10.1. Nulls, continues, and no-ops 51
4.10.2. Empty statements 51
4.10.3. Statements that instantiate generics 52
4.10.4. Begin…end and {…} pairs used as executable statements 52
4.10.5. Begin…end and {…} pairs that delimit (sub)program bodies 52
4.10.6. Logical expressions used as test conditions 53
4.10.7. Expression evaluations used as subprogram arguments 53
4.10.8. End symbols that terminate executable statements 53
4.10.9. End symbols that terminate declarations or (sub)program bodies 54
4.10.10. Then, else, and otherwise symbols 54
4.10.11. Elseif and elsif statements 54
4.10.12. Keywords like procedure division, interface, and implementation 54
4.10.13 Labels (branching destinations) on lines by themselves 54
4.11. Clarifications for Specific Languages 55
4.11.1. Stylistic use of braces 55
4.11.2. Expression statements 56
4.11.3. Format statements 56

**5. Using the Checklist to Define Size Measures—Nine Examples 59**
5.1. Physical Source Lines—A Basic Definition 60
5.2. Logical Source Statements—A Basic Definition 66
5.3. Data Specifications—Getting Data for Special Purposes 72
5.3.1. Specifying data for project tracking 72
5.3.2. Specifying data for project analysis 78
5.3.3. Specifying data for reuse measurement 82
5.3.4. Combining data specifications 85
5.3.5. Counting dead code 88

5.3.6. Relaxing the rule of mutual exclusivity 88
5.3.7. Asking for lower level details—subsets of attribute values 89
5.4. Other Definitions and Data Specifications 90
5.5. Other Measures 91

**6. Defining the Counting Unit 93**
6.1. Statement Delimiters 93
6.2. Logical Source Statements 93
6.3. Physical Source Lines 97
6.4. Physical and Logical Measures are Different Measures 99
6.5. Conclusion and Recommendation 101

**7. Reporting Other Rules and Practices—The Supplemental Forms 103**
7.1. Physical Source Lines 104
7.2. Logical Source Statements 105
7.3. Dead Code 106

**8. Recording Measurement Results 107**
8.1. Example 1—Development Planning and Tracking 108
8.2. Example 2—Planning and Estimating Future Products 109
8.3. Example 3—Reuse Tracking, Productivity Analysis, and Quality Normalization 110
8.4. Example 4—Support for Alternative Definitions and Recording Needs 112

**9. Reporting and Requesting Measurement Results 115**
9.1. Summarizing Totals for Individual Attributes 115
9.2. Summarizing Totals for Data Spec A (Project Tracking) 119
9.3. A Structured Process for Requesting Summary Reports 120
9.4. Requesting Arrayed Data 120
9.5. Requesting Individual (Marginal) Totals 126

**10. Meeting the Needs of Different Users 129**

**11. Applying Size Definitions to Builds, Versions, and Releases 133**

**12. Recommendations for Initiating Size Measurement 135**
12.1. Start with Physical Source Lines 135
12.2. Use a Checklist-Based Process to Define, Record, and Report Your Measurement Results 137
12.3. Report Separate Results for Questionable Elements 138
12.4. Look for Opportunities to Add Other Measures 138
12.5. Use the Same Definitions for Estimates that You Use for Measurement Reports 139
12.6. Dealing with Controversy 139
12.7. From Definition to Action—A Concluding Note 140

**References 141**

**Appendix A: Acronyms and Terms 143**
A.1. Acronyms 143
A.2. Terms Used 144

**Appendix B: Candidate Size Measures** **147**

**Appendix C: Selecting Measures for Definition—Narrowing the Field** **153**
C.1. The Choices Made 153
C.2. The Path Followed 155

**Appendix D: Using Size Measures—Illustrations and Examples** **157**
D.1. Project Tracking 158
D.2. Exposing Potential Cost Growth 159
D.3. Gaining Insight into Design and Coding Practices 160
D.4. Early Warnings: The Need for Schedule Replanning 162
D.5. Detecting Deferred Development 164
D.6. Indications of Design Turbulence 165
D.7. Normalizing Quality Measures 166

**Appendix E: Forms for Reproduction** **169**

# List of Figures

| | | |
|---|---|---|
| Figure 2-1 | Size Definition Framework—A Partial View | 6 |
| Figure 2-2 | Example of One Page of a Completed Checklist | 8 |
| Figure 2-3 | Application of Definition, Recording, and Reporting Forms to a Project | 11 |
| Figure 3-1 | A Data Array for Project Tracking | 14 |
| Figure 3-2 | Definition Checklist for Source Statement Counts | 16 |
| Figure 4-1 | The Statement Type Attribute | 24 |
| Figure 4-2 | The How Produced Attribute | 30 |
| Figure 4-3 | The Origin Attribute | 35 |
| Figure 4-4 | Relationships Between Origin and How Produced | 37 |
| Figure 4-5 | The Usage Attribute | 38 |
| Figure 4-6 | The Delivery Attribute | 39 |
| Figure 4-7 | The Functionality Attribute | 42 |
| Figure 4-8 | The Replications Attribute | 45 |
| Figure 4-9 | The Development Status Attribute | 48 |
| Figure 4-10 | The Language Attribute | 50 |
| Figure 4-11 | Clarifications—Inclusion and Statement Typing Rules for Program Elements | 50 |
| Figure 5-1 | Definition for Physical Source Lines of Code (SLOC) | 61 |
| Figure 5-2 | Definition for Logical Source Statements | 67 |
| Figure 5-3 | Example Data Array for Project Tracking | 73 |
| Figure 5-4 | Data Spec A (Project Tracking) | 74 |
| Figure 5-5 | The Case of Disappearing Reuse | 76 |
| Figure 5-6 | Specification for a Simplified Data Array | 77 |
| Figure 5-7 | Simplified Data Array for Project Tracking | 78 |
| Figure 5-8 | Example Data Array for Project Analysis | 79 |
| Figure 5-9 | Data Spec B (Project Analysis) | 80 |
| Figure 5-10 | Data Spec C (Reuse Measurement) | 83 |
| Figure 5-11 | Data Spec B+C (Combined Specifications) | 86 |
| Figure 5-12 | Dead Code Data Spec | 88 |
| Figure 5-13 | Specification for Counts of Textual Comments | 89 |
| Figure 5-14 | Getting Data on Subsets of Attribute Values | 90 |
| Figure 6-1 | Poll Results—Logical Source Statements | 96 |

Figure 6-2 Poll Results—Physical Source Lines 98
Figure 7-1 Rules Form—Counting Physical Source Lines 104
Figure 7-2 Rules Form—Counting Logical Source Statements 105
Figure 7-3 Practices Used to Identify Inoperative Elements 106
Figure 8-1 Recording Form for Data Spec A 108
Figure 8-2 Recording Form for Data Spec B 109
Figure 8-3 Recording Form for Data Spec C 111
Figure 8-4 A Generalized Form for Recording Measurements of Source Code Size 113
Figure 9-1 Example Summary Size Report 116
Figure 9-2 Summary Reporting Form for Data Spec A (Project Tracking) 119
Figure 9-3 Using Definition Checklists to Specify Special Information Needs 120
Figure 9-4 Request for a Report of Development Status vs. How Produced 122
Figure 9-5 Request for a Report of Statement Type vs. Programming Language 124
Figure 9-6 Request Specification for a Summary Report of Attribute Totals 127
Figure 10-1 Use of Checklists to Construct Common Definitions While Meeting the Needs of Different Measurement Users 130
Figure 10-2 Steps for Constructing and Using Definitions for Source Code Size 131
Figure 11-1 Applying Size Definitions to Builds, Versions, and Releases 134
Figure B-1 Software Size—A Partial Listing of Targets for Measurement 148
Figure B-2 Using Inputs and Outputs to Identify Activities 152
Figure C-1 Priorities for Definition 154
Figure C-2 The Path Followed When Applying the Framework to Construct Specific Definitions and Data Specifications 156
Figure D-1 Tracking Development Progress 158
Figure D-2 Exposing Potential Cost Growth—The Disappearance of Reused Code 159
Figure D-3 Comparison of Product Components—Uncovering Poor Design Practices 160
Figure D-4 Detecting Weak Designs—The Absence of Information Hiding 161
Figure D-5 Project Tracking—The Deviations May Seem Manageable 162

| | | |
|---|---|---|
| Figure D-6 | Project Tracking—Deviations from Original Plan Indicate Serious Problems | 163 |
| Figure D-7 | Project Tracking—Comparisons of Developer’s Plans Can Give Early Warnings of Problems | 163 |
| Figure D-8 | Indications of Deferred Development and Disappearing Reuse | 164 |
| Figure D-9 | Indications of Design Turbulence | 165 |
| Figure D-10 | Densities of Defects Discovered Before Products Were Released | 166 |
| Figure D-11 | Defect Densities for Firmware | 167 |

## Preface

In 1989, the Software Engineering Institute (SEI) began an effort to promote the use of measurement in the engineering, management, and acquisition of software systems. We believed that this was something that required participation from many members of the software community to be successful. As part of the effort, a steering committee was formed to provide technical guidance and to increase public awareness of the benefits of process and product measurements. Based on advice from the steering committee, two working groups were formed: one for software acquisition metrics and the other for software metrics definition. The first of these working groups was asked to identify a basic set of measures for use by government agencies that acquire software through contracted development efforts. The second was asked to construct measurement definitions and guidelines for organizations that produce or support software systems, and to give specific attention to measures of size, quality, effort, and schedule.

Since 1989, more than sixty representatives from industry, academia, and government have participated in SEI working group activities, and three resident affiliates have joined the Measurement Project staff. The Defense Advanced Research Projects Agency (DARPA) has also supported this work by making it a principal task under the Department of Defense Software Action Plan (SWAP). The results of these various efforts are presented here and in the following SEI reports:

- *Software Effort & Schedule Measurement: A Framework for Counting Staff-Hours and Reporting Schedule Information* (CMU/SEI-92-TR-21)
- *Software Quality Measurement: A Framework for Counting Problems and Defects* (CMU/SEI-92-TR-22)
- *Software Measures and the Capability Maturity Model* (CMU/SEI-92-TR-25)
- *Software Measurement Concepts for Acquisition Program Managers* (CMU/SEI-92-TR-11)
- *A Concept Study for a National Software Engineering Database* (CMU/SEI-92-TR-23)
- *Software Measurement for DoD Systems: Recommendations for Initial Core Measures* (CMU/SEI-92-TR-19)

This report and the methods in it are outgrowths of work initiated by the Size Subgroup of the Software Metrics Definition Working Group. Like the reports listed above, this one contains guidelines and advice from software professionals. It is not a standard, and it should not be viewed as such. Nevertheless, the framework and methods it presents give a solid basis for constructing and communicating clear definitions for two important measures that can help us plan, manage, and improve our software projects and processes.

We hope that the materials we have assembled will give you a solid foundation for making your size measures repeatable, internally consistent, and clearly understood by others. We also hope that some of you will take the ideas illustrated in this report and apply them to other measures, for no single set of measures can ever encompass all that we need to know about software products and processes.

Our plans at the SEI are to continue our work in software process measurement. If, as you use this report, you discover ways to improve its contents, please let us know. We are especially interested in lessons learned from operational use that will help us improve the advice we offer to others. With sufficient feedback, we may be able to refine our work or publish additional useful materials on software size measurement.

Our point of contact for comments is

Kathy Cauley
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

# Acknowledgments

The SEI measurement efforts depend on the participation of many people. We would like to thank the members of the Size Subgroup of the Software Metrics Definition Working Group who contributed to the preparation of this report. The SEI is indebted to them and to the organizations who sponsored their efforts to improve the measurement of software size. Without the contributions of these professionals, we could not have completed this task:

Jeffrey Aune
Paramax

Charles Cox
Naval Weapons Center

Paul Funch
The MITRE Corporation

John Gaffney
Software Productivity Consortium

Nancy Jensen
Rockwell International

Raymond Kile
Hughes Aircraft Company

Charles Koch
Naval Air Development Center

Walter Lamia
Software Engineering Institute

Shari Lawrence Pfleeger
The MITRE Corporation

Stan Rifkin
Master Systems, Inc.

David Seaver
Project Engineering, Inc.

Edward Tilford, Sr.
Paramax

A first draft of this report was presented and distributed for review at the SEI Affiliates Symposium in August 1991. A second draft was distributed to approximately 400 reviewers in June 1992. More than 200 comments and suggestions for improvement were returned. All have received careful consideration, and most have been either incorporated or addressed through the development of new materials. We are indebted to those who took the time and care to provide so many constructive recommendations:

Julia Allen
Software Engineering Institute

Jim Bartlett
Allstate Insurance Company

Barry Boehm
Defense Advanced Research
Projects Agency

John Bolland
ITT Avionics

Lyle Cocking
General Dynamics

Thomas Donovan
Defense Logistics Agency

Dean Dubofsky
The MITRE Corporation

Betty Falato
Federal Aviation Administration

| | |
|---|---|
| M. Hosein Fallah<br>AT&T Bell Laboratories | Karen Powel<br>McDonnell Douglas |
| Liz Flanagan<br>AT&T Bell Laboratories | Donald Reifer<br>Reifer Consultants, Inc. |
| Harvey Hallman<br>Software Engineering Institute | Paul Rook<br>S.E.P.M. |
| James Hart<br>Software Engineering Institute | William Rooney<br>AIL Systems Inc. |
| George Huyler<br>Productivity Management Group, Inc. | John Salasin<br>Software Engineering Institute |
| Chris Kemerer<br>Massachusetts Institute of Technology | Hal Schwartz<br>Fujitsu Systems of America |
| Gary Kennedy<br>IBM Corporation | Brian Sharpe<br>Hewlett-Packard |
| Harry Larson<br>Larbridge Enterprises | Marie Silverthorn<br>Texas Instruments |
| Donna Lindskog<br>University of Regina and SaskTel | Al Snow<br>AT&T Bell Laboratories |
| Marc Meltzer<br>Pratt & Whitney | S. Jack Sterling<br>Logicon Eagle Technology, Inc. |
| Everald Mills<br>Seattle University | Irene Stone<br>AIL Systems, Inc. |
| Kerux-David Lee Neal<br>Northrop Corporation | Terry Wilcox<br>DPRO-General Dynamics |

We also thank the members of the Measurement Steering Committee for their many thoughtful contributions. The insight and advice they have provided have been invaluable. This committee consists of senior representatives from industry, government, and academia who have earned solid national and international reputations for their contributions to measurement and software management:

William Agresti
The MITRE Corporation

Henry Block
University of Pittsburgh

David Card
Computer Sciences Corporation

Andrew Chruscicki
USAF Rome Laboratory

Samuel Conte
Purdue University

Bill Curtis
Software Engineering Institute

Joseph Dean
Tecolote Research

Stewart Fenick
US Army Communications-Electronics Command

Charles Fuller
Air Force Materiel Command

Robert Grady
Hewlett-Packard

John Harding
Bull HN Information Systems, Inc.

Frank McGarry
NASA (Goddard Space Flight Center)

John McGarry
Naval Underwater Systems Center

Watts Humphrey
Software Engineering Institute

Richard Mitchell
Naval Air Development Center

John Musa
AT&T Bell Laboratories

Alfred Peschel
TRW

Marshall Potter
Department of the Navy

Samuel Redwine
Software Productivity Consortium

Kyle Rone
IBM Corporation

Norman Schneidewind
Naval Postgraduate School

Herman Schultz
The MITRE Corporation

Seward (Ed) Smith
IBM Corporation

Robert Sulgrove
NCR Corporation

Ray Wolverton
Hughes Aircraft

As we prepared this report, we were aided in our activities by the able and professional support staff of the SEI. Special thanks are owed to Mary Beth Chrissis and Suzanne Couturiaux, who were instrumental in getting our early drafts ready for external review; to Linda Pesante and Mary Zoys, whose editorial assistance helped guide us to a final, publishable form; to Marcia Theoret and Lori Race, who coordinated our meeting activities and provided outstanding secretarial services; and to Helen Joyce and her assistants, who so competently assured that meeting rooms, lodgings, and refreshments were there when we needed them.

And finally, we could not have assembled this report without the active participation and contributions from the other members of the SEI Software Process Measurement Project and the SWAP team who helped us shape these materials into forms that could be used by both industry and government practitioners:

Anita Carleton
Software Engineering Institute

John Baumert
Computer Sciences Corporation

Mary Busby
IBM Corporation

Elizabeth Bailey
Institute for Defense Analyses

Andrew Chruscicki
USAF Rome Laboratory

Judith Clapp
The MITRE Corporation

William Florac
Software Engineering Institute

Wolfhart Goethert
Software Engineering Institute

Donald McAndrews
Software Engineering Institute

Shari Lawrence Pfleeger
The MITRE Corporation

Lori Race
Software Engineering Institute

James Rozum
Software Engineering Institute

Timothy Shimeall
Naval Postgraduate School

Patricia Van Verth
Canisius College

# Software Size Measurement: A Framework for Counting Source Statements

**Abstract**. This report presents guidelines for defining, recording, and reporting two frequently used measures of software size—physical source lines and logical source statements. We propose a general framework for constructing size definitions and use it to derive operational methods for reducing misunderstandings in measurement results. We show how the methods can be applied to address the information needs of different users while maintaining a common definition of software size.

# 1. Introduction

## 1.1. Scope

Size measures have direct application to the planning, tracking, and estimating of software projects. They are used also to compute productivities, to normalize quality indicators, and to derive measures for memory utilization and test coverage.

This report presents methods for reducing misunderstandings and inconsistencies when recording and reporting measures of software size. In it we provide the following:

- A framework for constructing and communicating definitions of size.
- A checklist for defining and describing the coverage of two frequently used measures—source lines of code (SLOC) and logical source statements.
- Example definitions for counts of physical source lines and logical source statements.
- Checklists for requesting and specifying additional data elements useful for project planning and tracking.
- Forms for recording and reporting measurement results.
- Examples of uses and interpretations of software size measures.
- Recommendations for implementation.

This report does not address measures of difficulty, complexity, or other product and environmental characteristics that we often use when interpreting measures of software size. These measures are themselves important subjects for definition. Each deserves to be defined with the thoroughness we try to apply here to measures of size. If our methods become viewed as useful, we hope that they will inspire future work aimed at developing procedures for describing and communicating explicit definitions for these and other important software characteristics.

## 1.2. Objective and Audience

Our objective is to provide operational methods that will help organizations obtain clear and consistent reports of software size. Managers, developers, maintainers, estimators, and process improvement teams should all find our methods helpful in getting the data they need to plan, control, and improve the processes they use.

The goal of our methods is to reduce ambiguities and misunderstandings in reported measures of size. Our motivation is that we see far too many reports of size that convey little useful information. Moreover, reported values for software size are often confusing and easily misinterpreted. This usually happens because neither the conveyors nor the receivers of the information know what the measurements include or whether the measures have been applied with any consistency. This confusion becomes compounded when those making the report do not themselves know what actions were taken to collect the measurements they cite. As a consequence, reports like "Our software activity produced 163,000 source code instructions on that job" can easily be misunderstood by a factor of three or more. Ambiguous reports like these place activities like project estimating, project management, and process improvement on shaky foundations.

To remove these ambiguities, we have developed a framework that helps us describe software size measurements in ways that are both complete and explicitly communicated. We have applied this framework to construct examples of specifications for collecting and reporting two measures of source code size—physical lines and logical source statements

Good definitions and good specifications require attention to detail. When you first use the methods in this report to construct definitions, the number of issues you will address may seem formidable. But decisions on most of these issues need be made only once—thereafter they become organizational standards. The full details provided by the methods in the report will then become especially valuable to the tool builders and corporate metrics specialists who are charged with placing your organization's software measurements on a sound foundation.

## 1.3. Relationship to Other Work

The benefits of software measurements are recognized more widely today than ever before. Several recent publications illustrate the kinds of quantitative indicators that organizations could be using to plan, control, and improve their software processes. We provide examples of some of these indicators in Appendix D, and others can be found in the MITRE metrics [Schultz 88], the proposed Army STEP metrics [Betz 92], the Air Force management indicators [USAF 92], and in publications like [Baumert 92], [Grady 87], [McGhan 91], [McGhan 92], and [Rozum 92].

While these publications show many graphs that we can use to help plan and track software products and processes, few define the underlying measures on which their indicators are based. Without this information, it is easy for incorrect conclusions to be drawn and inappropriate decisions to be made.

This report provides a framework for defining some of the fundamental size measures used in software indicators. It also provides methods for ensuring that rules of measurement remain consistent, so that reports made tomorrow can be related to those made today. You can use these methods also to achieve common definitions across projects and across organizations, so that long term trends can be tracked and lessons learned from one project or organization can be extrapolated to another.

## 1.4. Endorsement of Specific Measures

Nothing in this report should be interpreted as implying that we believe one size measure is more useful or more informative than another. To the best of our knowledge, no one has yet used the kind of measurement definitions that would permit conclusions of this sort to be validated. Tools like the checklists and forms in this report should help all of us begin to assemble the information that will let us determine which size measures are the most effective for the different processes we use in developing and supporting software systems.

In the meantime, we do believe that we can sometimes judge which measures will be easiest to implement and which will be easiest to automate. Where we have elected to pursue one measure rather than another, these judgments have played major roles. The fact that we choose or recommend a particular path in this report means only that we judge it to be the best course to take today to get immediately usable results.

# 2. A Framework for Size Definition

The measurement framework we propose in this report makes extensive use of checklists. These checklists provide operational methods for constructing and communicating clearly understood definitions of software size measures (our primary objective). They also permit us to satisfy the data needs of different users while maintaining consistent definitions of size. This lets us expand and tailor our measurement data sets as our process maturities increase, without having to change our underlying definitions or measurement practices.

Before defining any measure, we should always ask "Why do we want the information we propose to collect, and how will we use it?" The answers to these questions become our measurement objectives, and each objective will have its own coverage requirements and information needs. Checklists like those in this report give us a means for addressing and defining the rules that we follow to meet these needs.

In developing our framework and checklists, we have been guided by two principal criteria:

- *Communication:* If someone uses our methods to define a measure or describe a measurement result, will others know precisely what has been measured and what has been included and excluded?
- *Repeatability:* Would someone else be able to repeat the measurement and get the same result?

The framework we use consists of the following steps, which we apply to each prospective size measure:

1. Identify the principal attributes that characterize the objects we want to measure.
2. Identify the values of each attribute that different users of the measure may want to either include in or exclude from their measurement results. Ensure that these values are mutually exclusive.
3. Prepare a checklist of principal attributes and their values, so that values included in and excluded from measures can be explicitly identified.
4. Ensure that we understand why coverage or visibility of each attribute value may be important to one or more users of measurement results.
5. For each attribute, identify the values we will include in the measure, together with those we will exclude. Record the inclusions and exclusions on the checklist.
6. Identify and record all additional rules and exceptions that we follow when collecting and recording measurement results. Use the results of this step, in conjunction with those of step 5, as the definition for size.
7. Identify and record the values (or sets of values) for which individual measurements will be collected and recorded. Use the results of this step as data specifications.
8. Make and record measurements according to the definition and data specifications.

9. Aggregate the measurement results and prepare summary reports.
10. Attach the measurement definition and data specifications to each set of measurement records and reports.

These criteria and steps have led us to four kinds of instruments for defining, recording, and reporting software size measurements—definition checklists, supplemental rules forms, data recording forms, and reporting forms. The definition checklists, in turn, have at least three uses. We use them to define our overall measure for size, to create specifications for recording additional data elements (data specifications), and to convey user requests for specialized reports (data requests). Figure 2-1 illustrates the overall scheme. We describe the forms we use in Chapters 3 through 9 and present reproducible copies in Appendix E.

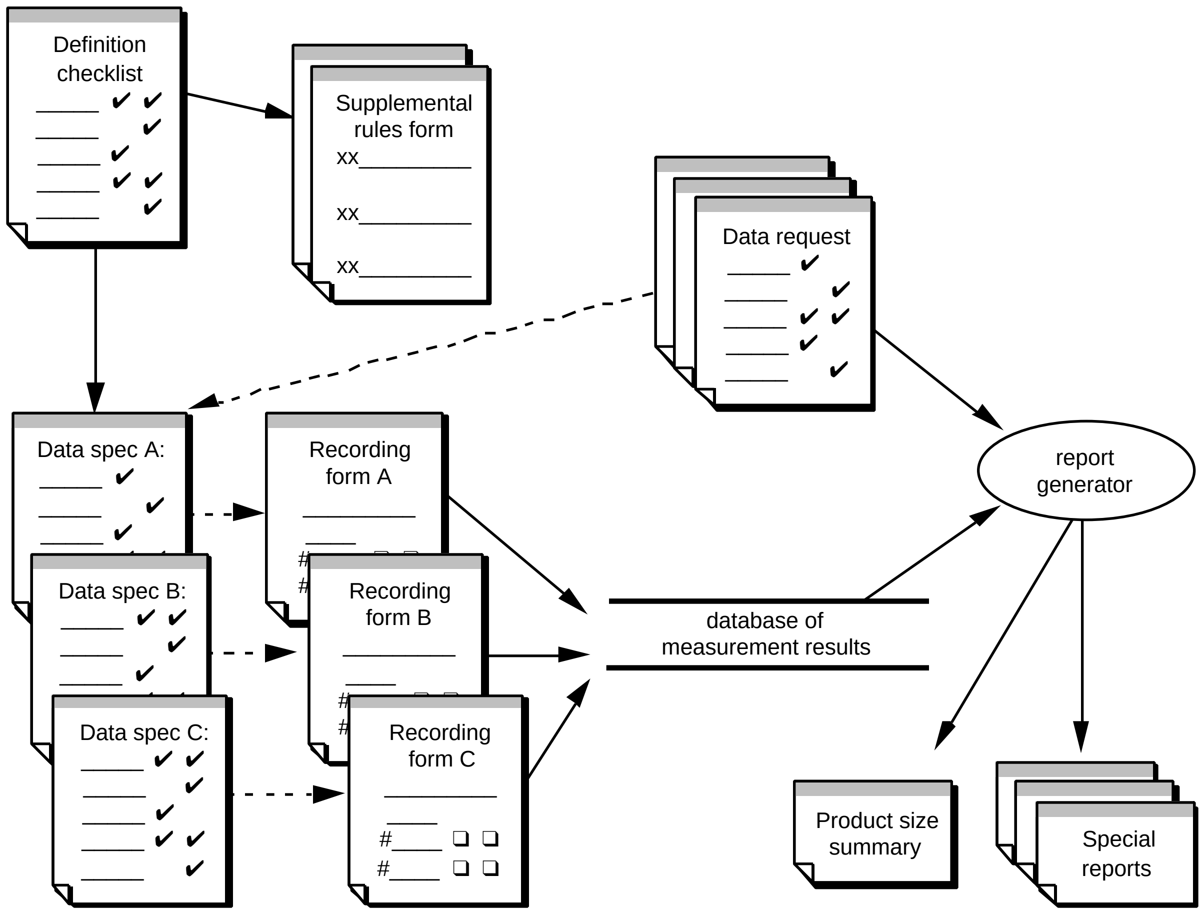


Figure 2-1 Size Definition Framework—A Partial View

## 2.1. Definition Checklists

The methods we propose for constructing size definitions start with a checklist. The checklist identifies the principal attributes of our target size measure. Principal attributes are characteristics important to the different people who will use the measurement results. Examples of attributes of familiar objects include properties such as length, weight, height, volume, density, color, source language, location, origin, destination, function performed, author, and status. For each attribute of our target measure, we identify the values that the attribute can take on. In each case, these values should be both exhaustive (complete) and mutually exclusive (nonoverlapping).

After listing principal attributes and their values and arranging them into a checklist, the process for constructing a definition becomes relatively straightforward. We simply check off the attribute values we will include in our definition and exclude all others. We also design and use supporting forms to record special rules and clarifications that are not amenable to checklist treatment.

The checklist has other uses as well. The format we use permits us to designate the data elements associated with the definition that we want to have measured and recorded when measurements are made. This lets us convey our needs to those who make the measurements and those who set up and maintain the databases in which measurement results are stored. The checklist can also be used to request and specify the data arrays and marginal (individual) totals that we would like to have reported to us.

In practice, a checklist turns out to be a very flexible tool. For example, an organization may want to merge results from several values into a new value. Moreover, some measures exist (counts of source code comments, for example) that some organizations may want to record and aggregate but not include in a total size measure. All these options can be addressed with the checklist.

With this in mind, we have found it useful to provide blank lines in checklists so that organizations can add other attribute values to meet local needs. When you exercise this flexibility to list additional values for inclusion or exclusion, you should take care to rephrase the labels for existing values so that overlaps do not occur.

The importance of ensuring that values within attributes are nonoverlapping cannot be overstated. If values are not mutually exclusive, observed results can get assigned to more than one category. If this happens, and if totals (such as total size) are computed by adding across all values within an attribute that are designated for inclusion, double counting can occur. Reported results will then be larger than they really are. In the same vein, if overlaps exist between two values and if one of the values is included within a definition while the other is not, those who collect the data will not know what to do with observations that fall into both classes.

## Definition Checklist for Source Statement Counts

Definition name: ***Physical Source Lines of Code (basic definition)*** Date: ***8/7/92***

Originator: ***SEI***

**Measurement unit:** **Physical source lines** ✓ **Logical source statements** ☐

| **Statement type** — **Definition** ✓ **Data array** ☐ | | **Includes** | **Excludes** |
|---|---|---|---|
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | |
| 1 Executable **Order of precedence ->** | 1 | ✓ | |
| 2 Nonexecutable | | | |
| 3 Declarations | 2 | ✓ | |
| 4 Compiler directives | 3 | ✓ | |
| 5 Comments | | | |
| 6 On their own lines | 4 | | ✓ |
| 7 On lines with source code | 5 | | ✓ |
| 8 Banners and nonblank spacers | 6 | | ✓ |
| 9 Blank (empty) comments | 7 | | ✓ |
| 10 Blank lines | 8 | | ✓ |
| 11 | | | |
| 12 | | | |

| **How produced** — **Definition** ✓ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|
| 1 Programmed | ✓ | |
| 2 Generated with source code generators | ✓ | |
| 3 Converted with automated translators | ✓ | |
| 4 Copied or reused without change | ✓ | |
| 5 Modified | ✓ | |
| 6 Removed | | ✓ |
| 7 | | |
| 8 | | |

| **Origin** — **Definition** ✓ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|
| 1 New work: no prior existence | ✓ | |
| 2 Prior work: taken or adapted from | | |
| 3 A previous version, build, or release | ✓ | |
| 4 Commercial, off-the-shelf software (COTS), other than libraries | ✓ | |
| 5 Government furnished software (GFS), other than reuse libraries | ✓ | |
| 6 Another product | ✓ | |
| 7 A vendor-supplied language support library (unmodified) | | ✓ |
| 8 A vendor-supplied operating system or utility (unmodified) | | ✓ |
| 9 A local or modified language support library or operating system | ✓ | |
| 10 Other commercial library | ✓ | |
| 11 A reuse library (software designed for reuse) | ✓ | |
| 12 Other software component or library | ✓ | |
| 13 | | |
| 14 | | |

| **Usage** — **Definition** ✓ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|
| 1 In or as part of the primary product | ✓ | |
| 2 External to or in support of the primary product | | ✓ |
| 3 | | |

Figure 2-2 Example of One Page of a Completed Checklist

Figure 2-2 is an example of how the first page of a completed checklist might look for one particular definition of software size. Bold-faced section headings highlight the attributes, and these headings are followed by the lists of values that the attributes take on. For example, **Statement type** is an attribute, and the values for this attribute are executable statements, declarations, compiler directives, comments, and blank lines. All except the first have been grouped under a generic class called nonexecutable, for which some organizations may want aggregated accountings. Chapter 3 presents the full checklist, and Chapter 4 discusses the attributes and their values.

Checklists like the one in Figure 2-2 help provide a structured approach for dealing with the details that must be resolved to reduce misunderstandings when collecting and communicating measures of software size. With such checklists, issues can be stepped through and addressed one at a time by designating the elements that people want included in measurement results. At the same time, designating the elements to be excluded directs attention to actions that must be taken to avoid contaminating measurement results with unwanted elements.

In Chapter 5, we will present examples of two definitions that we have constructed with this checklist. Although these examples represent consensus views, readers should keep in mind that there are no universal “best” choices when completing a definition checklist. Instead, each choice should be made so as to serve an organization’s overall measurement needs. This almost always involves tradeoffs between the benefits to be gained (based on how the measurement results will be used) and the difficulties associated with applying the definition to collect data from real software projects.

## 2.2. Supplemental Rules Forms

Sometimes definition checklists cannot explain all the rules and practices that must be made explicit to avoid ambiguities and misunderstandings. In these instances, we recommend constructing specialized forms to spell out the additional rules. There are three instances where we have found this necessary:

- When defining rules for distinguishing among different statement types while counting physical source lines.
- When defining the delimiters and rules used to identify beginnings and endings of different kinds of logical source statements.
- When describing procedures used to find and identify dead code, so that only operative software is included in system size.

Chapter 7 (Figures 7-1, 7-2, and 7-3) presents the forms we have created for describing these practices.

When completed checklists cannot provide full disclosure of all measurement rules, they should be accompanied by supplemental rules forms. All entries on the supplemental forms should be either filled in or marked as not applicable, so that no loose ends are left hanging.

The combination of a completed checklist and its supplemental rules forms becomes a vehicle for communicating the meaning of measurement results to others. The checklist and supplemental forms can be used for this purpose whether or not definitions have been agreed to in advance. They can also be used at the start of a project to negotiate and establish standards for collecting and reporting measures of software size. The benefits become even greater when the standards are applied uniformly across multiple projects and organizations.

## 2.3. Recording and Reporting Forms

Although checklists and rules forms are useful for recording and communicating definitions, they neither record nor communicate results. For definitions to work, they must be supported by forms that people can use to collect and transmit results of quantitative observations. There are two steps in this process—recording and reporting. Our framework distinguishes between these steps and provides forms for each.

**Recording forms.** The purpose of recording forms is to transport data from those who make measurements to those who enter the results into databases. These are low-level forms, designed for measuring relatively small and homogeneous units. Recording forms should be consistent with the data elements designated for measurement, and they should capture all information needed to track the data back both to the definition and to the entity measured. They should also include provisions for noting the time and stage within the life cycle or process where the measurements are made. Chapter 8 presents examples of forms we have constructed for recording counts of physical and logical source statements.

**Reporting forms.** The purpose of reporting forms is to aggregate and summarize data for those who use measurement results. Our ability to do this effectively in a multidimensional world is constrained severely by the two-dimensional nature of most presentation media. We will present an example of a summary report and a process for specifically addressing the data needs of individual users in Chapter 9.

Figure 2-3 illustrates the relationships among the various forms when several languages are present. Here, a definition checklist for logical source statements is supported by rules forms that explain the exact practices employed when identifying statements in different programming languages. These forms are supported in turn by recording forms for collecting data and reporting forms for summarizing measurement results. Separate copies of the recording forms are used for each software entity measured. Thus, several (or even many) recording forms may be completed within a given project.

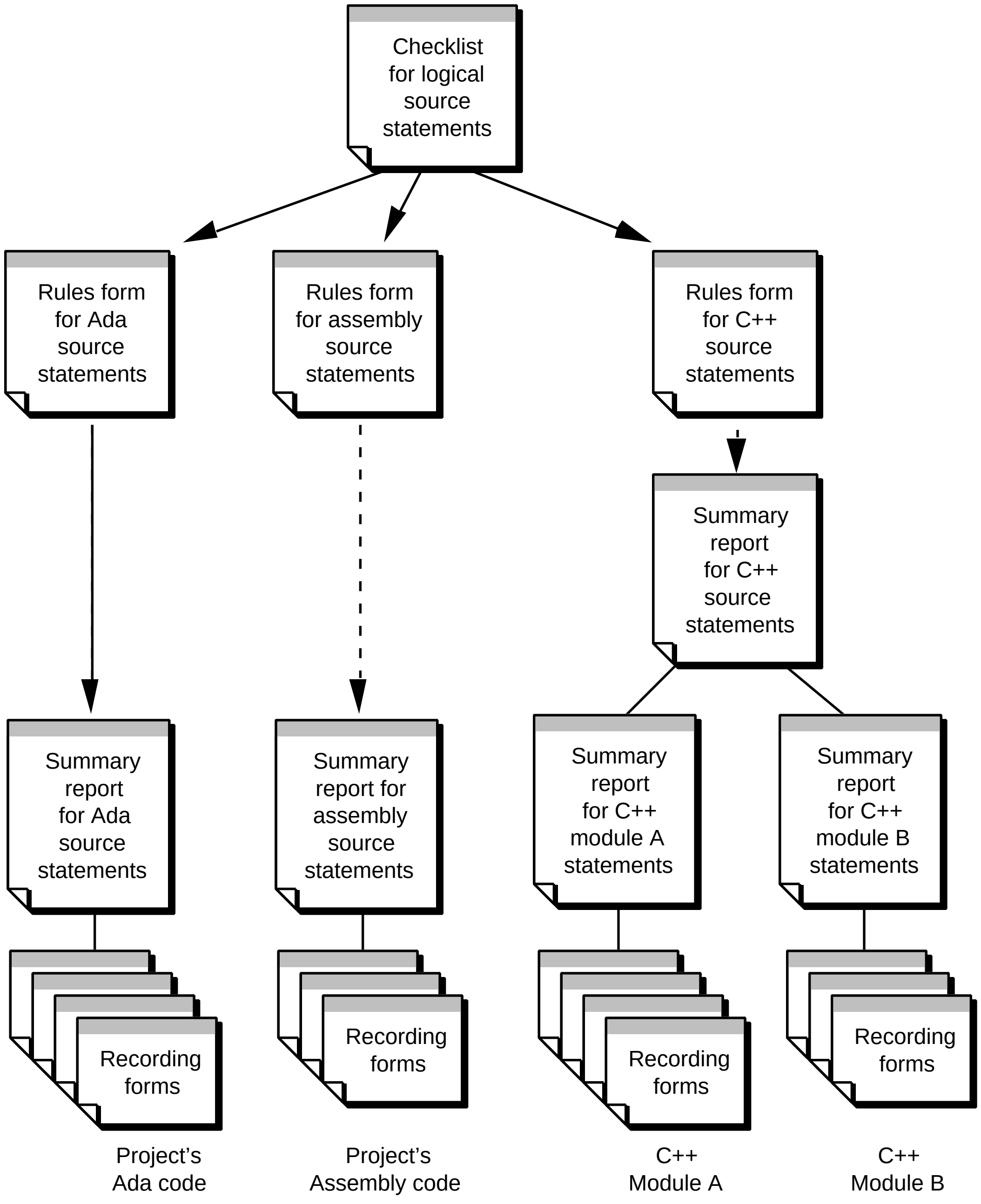

Checklist for logical source statements
Rules form for Ada source statements
Rules form for assembly source statements
Rules form for C++ source statements
Summary report for C++ source statements
Summary report for Ada source statements
Summary report for assembly source statements
Summary report for C++ module A statements
Summary report for C++ module B statements
Recording forms
Recording forms
Recording forms
Recording forms
Project's Ada code
Project's Assembly code
C++ Module A
C++ Module B

Figure 2-3 Application of Definition, Recording, and Reporting Forms to a Project

# 3. A Checklist for Defining Source Statement Counts

In this chapter we present a checklist for defining two measures of source code size—physical source lines and logical source statements. Counts of physical lines describe size in terms of the physical length of the code as it appears when printed for people to read. Counts of logical statements, on the other hand, attempt to characterize size in terms of the number of software instructions, irrespective of their relationship to the physical formats in which they appear.

Some common acronyms for physical source lines are LOC, SLOC, KLOC, KSLOC, and DSLOC, where SLOC stands for source lines of code, K (kilo) indicates that the scale is in thousands, and D says that only delivered source lines are included in the measure. Common abbreviations for logical source statements include LSS, DSI, and KDSI, where DSI stands for delivered source instructions.

In addressing these two measures of software size, we have drawn extensively from the draft *Standard for Software Productivity Metrics* prepared by the P1045 Working Group of the IEEE [IEEE 92]. This draft standard provides many terms and definitions that are useful in arriving at mutually understood measures of software productivity. We were fortunate to have the evolving standard available to us during our work.

However, perhaps because the IEEE work is directed more toward derived measures (productivity = outputs divided by inputs) than toward size itself, it sometimes stops short of the detailed issues that must be settled if ambiguities in source code counts are to be avoided. Size measures can be used for more than just productivity computations. For example, counts of physical and logical statements can be very useful for cost estimating, for tracking the progress of projects with respect to plans, for monitoring the development of reuse libraries, and for evaluating the effectiveness of reuse strategies. They can also be practical bases for normalizing other software measures, such as those used to describe quality (defect densities) and product improvement efforts (rates of error discovery and efficiency of peer reviews).

Thus, there are reasons to probe more deeply than the P1045 Working Group has probed. The checklist we have developed provides a structured method for extending and refining the size definition efforts initiated by the IEEE. For the most part, it is consistent with their work. To help maintain this consistency, we have used the term *source statements* to refer to both physical source lines and logical source statements.

## 3.1. Describing Statement Counts

Logical statements and physical lines of code are characterized by their attributes. In fact, it is only through the values of attributes that size measures have any definition or meaning. Historically, the primary problem with measures of source code size has not been in coming up with numbers—anyone can do that. Rather, it has been in identifying and communicating the attributes that describe exactly what those numbers represent. If we do not identify and describe the attributes of size, we cannot guarantee the consistency needed to make size measurements useful.

The checklist is primarily a tool for identifying and addressing attributes. Figure 3-2 shows the one we have created for defining source statement counts. This checklist may, at first glance, seem complicated. Initial reactions are often of the sort: “Good grief! Do I really have to wrestle with all that detail?” We can only observe that if you do not, others can (and often will) do anything they want with respect to issues left unaddressed. The consequence is that the value of collected data is diminished and its meaning becomes corrupted. Unfortunately, this is too often characteristic of software size measurement today.

The definition checklist in Figure 3-2 gives a mechanism for communicating exactly what has been included in—and what has been excluded from—counts of source code size. It can be used to record the rules used for counting either source lines of code (a physical measure) or logical source statements (instructions). It can be used also to construct and communicate requests and specifications for collecting and recording counts of individual data elements. Since attributes take on values independent of each other, data on these individual elements will usually be collected in arrays. Figure 3-1 is an example of an array that reports statement counts for individual data elements.

|  | coded | unit tested | integrated | system tests completed | total |
|---|---|---|---|---|---|
| programmed | 20,224 | 33,731 | 16,432 | 0 | 70,387 |
| copied | 5,108 | 10,883 | 18,631 | 0 | 34,622 |
| modified | 3,006 | 4,865 | 5,194 | 0 | 13,065 |
| total | 28,338 | 49,479 | 40,257 | 0 | 118,074 |

Figure 3-1 A Data Array for Project Tracking

In Chapter 5, we will use the **Definition** and **Data array** boxes of the checklist to construct some specific examples of definitions and data array specifications. In the meantime, users can put the checklist to immediate use by using it to describe their current measures of software size. For this purpose, you should check the **Definition** boxes if you are not using the checklist to describe arrayed data.

Since the first purpose of the checklist is to describe measures that have already been made, it can be used with very little explanation by any organization that has been collecting and reporting counts of either physical source lines or logical source statements. The focus here is on communication. Users need only check off each element on the checklist to indicate whether or not it is included in or excluded from their measurement results. The completed checklist can then be attached to the reported results, so that users of the results will know what the numerical values represent. The same process can be used also to describe the software sizes used for estimating project costs and for project planning and scheduling.

The checklist in Figure 3-2 uses nine attributes to describe and bound the kinds of software statements included in a measure of source code size. The attributes are: **Statement type**, **How produced**, **Origin**, **Usage**, **Delivery**, **Functionality**, **Replications**, **Development status**, and **Language**. These attributes are orthogonal, or nearly so, in the sense that each takes on values more or less independently of the others. Values within an attribute do not overlap, and each represents a class of statements that is of interest to one or more of the software or management communities that use the results of size measurement. When reporting a result or creating a definition, users have only to check the attribute values they include and those they exclude when measuring and reporting counts of source statements.

When using the checklist, you are always free to collect the underlying data at any level of granularity you wish. The checklist is merely a vehicle for reporting the rules applied when assembling the data for others to use.

## Definition Checklist for Source Statement Counts

Definition name: ______________________________ Date: __________

______________________________ Originator: __________

| | | | | |
|---|---|---|---|---|
| **Measurement unit:** | **Physical source lines** ☐ | | | |
| | **Logical source statements** ☐ | | | |
| **Statement type** | **Definition** ☐ **Data array** ☐ | | **Includes** | **Excludes** |
| | *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | |
| 1 | Executable | **Order of precedence ->** 1 | | |
| 2 | Nonexecutable | | | |
| 3 | Declarations | 2 | | |
| 4 | Compiler directives | 3 | | |
| 5 | Comments | | | |
| 6 | On their own lines | 4 | | |
| 7 | On lines with source code | 5 | | |
| 8 | Banners and nonblank spacers | 6 | | |
| 9 | Blank (empty) comments | 7 | | |
| 10 | Blank lines | 8 | | |
| 11 | | | | |
| 12 | | | | |
| **How produced** | **Definition** ☐ **Data array** ☐ | | **Includes** | **Excludes** |
| 1 | Programmed | | | |
| 2 | Generated with source code generators | | | |
| 3 | Converted with automated translators | | | |
| 4 | Copied or reused without change | | | |
| 5 | Modified | | | |
| 6 | Removed | | | |
| 7 | | | | |
| 8 | | | | |
| **Origin** | **Definition** ☐ **Data array** ☐ | | **Includes** | **Excludes** |
| 1 | New work: no prior existence | | | |
| 2 | Prior work: taken or adapted from | | | |
| 3 | A previous version, build, or release | | | |
| 4 | Commercial, off-the-shelf software (COTS), other than libraries | | | |
| 5 | Government furnished software (GFS), other than reuse libraries | | | |
| 6 | Another product | | | |
| 7 | A vendor-supplied language support library (unmodified) | | | |
| 8 | A vendor-supplied operating system or utility (unmodified) | | | |
| 9 | A local or modified language support library or operating system | | | |
| 10 | Other commercial library | | | |
| 11 | A reuse library (software designed for reuse) | | | |
| 12 | Other software component or library | | | |
| 13 | | | | |
| 14 | | | | |
| **Usage** | **Definition** ☐ **Data array** ☐ | | **Includes** | **Excludes** |
| 1 | In or as part of the primary product | | | |
| 2 | External to or in support of the primary product | | | |
| 3 | | | | |

Figure 3-2 Definition Checklist for Source Statement Counts

Definition name: ______________________________

______________________________

| | | | | | |
|---|---|---|---|---|---|
| **Delivery** | | **Definition** ☐ | **Data array** ☐ | **Includes** | **Excludes** |
| 1 | Delivered | | | ▨ | ▨ |
| 2 | Delivered as source | | | | |
| 3 | Delivered in compiled or executable form, but not as source | | | | |
| 4 | Not delivered | | | ▨ | ▨ |
| 5 | Under configuration control | | | | |
| 6 | Not under configuration control | | | | |
| 7 | | | | | |
| **Functionality** | | **Definition** ☐ | **Data array** ☐ | **Includes** | **Excludes** |
| 1 | Operative | | | | |
| 2 | Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | | ▨ | ▨ |
| 3 | Functional (intentional dead code, reactivated for special purposes) | | | | |
| 4 | Nonfunctional (unintentionally present) | | | | |
| 5 | | | | | |
| 6 | | | | | |
| **Replications** | | **Definition** ☐ | **Data array** ☐ | **Includes** | **Excludes** |
| 1 | Master source statements (originals) | | | | |
| 2 | Physical replicates of master statements, stored in the master code | | | | |
| 3 | Copies inserted, instantiated, or expanded when compiling or linking | | | | |
| 4 | Postproduction replicates—as in distributed, redundant, or reparameterized systems | | | | |
| 5 | | | | | |
| **Development status** | | **Definition** ☐ | **Data array** ☐ | **Includes** | **Excludes** |
| | *Each statement has one and only one status, usually that of its parent unit.* | | | ▨ | ▨ |
| 1 | Estimated or planned | | | | |
| 2 | Designed | | | | |
| 3 | Coded | | | | |
| 4 | Unit tests completed | | | | |
| 5 | Integrated into components | | | | |
| 6 | Test readiness review completed | | | | |
| 7 | Software (CSCI) tests completed | | | | |
| 8 | System tests completed | | | | |
| 9 | | | | | |
| 10 | | | | | |
| 11 | | | | | |
| **Language** | | **Definition** ☐ | **Data array** ☐ | **Includes** | **Excludes** |
| | *List each source language on a separate line.* | | | ▨ | ▨ |
| 1 | | ______________ | | | |
| 2 | Job control languages | ______________ | | | |
| 3 | | ______________ | | | |
| 4 | Assembly languages | ______________ | | | |
| 5 | | ______________ | | | |
| 6 | Third generation languages | ______________ | | | |
| 7 | | ______________ | | | |
| 8 | Fourth generation languages | ______________ | | | |
| 9 | | ______________ | | | |
| 10 | Microcode | ______________ | | | |
| 11 | | | | | |

Figure 3-2 Definition Checklist for Source Statement Counts, Page 2

| Definition name: ______________________ | | Includes | Excludes |
|---|---|---|---|
| **Clarifications (general)** **Listed elements are assigned to statement type –>** | | | |
| 1 Nulls, continues, and no-ops | | | |
| 2 Empty statements (e.g., “;;” and lone semicolons on separate lines) | | | |
| 3 Statements that instantiate generics | | | |
| 4 Begin…end and {…} pairs used as executable statements | | | |
| 5 Begin…end and {…} pairs that delimit (sub)program bodies | | | |
| 6 Logical expressions used as test conditions | | | |
| 7 Expression evaluations used as subprogram arguments | | | |
| 8 End symbols that terminate executable statements | | | |
| 9 End symbols that terminate declarations or (sub)program bodies | | | |
| 10 Then, else, and otherwise symbols | | | |
| 11 Elseif statements | | | |
| 12 Keywords like procedure division, interface, and implementation | | | |
| 13 Labels (branching destinations) on lines by themselves | | | |
| 14 | | | |
| 15 | | | |
| 16 | | | |
| **Clarifications (language specific)** | | | |
| **Ada** | | | |
| 1 End symbols that terminate declarations or (sub)program bodies | | | |
| 2 Block statements (e.g., begin…end) | | | |
| 3 With and use clauses | | | |
| 4 When (the keyword preceding executable statements) | | | |
| 5 Exception (the keyword, used as a frame header) | | | |
| 6 Pragmas | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **Assembly** | | | |
| 1 Macro calls | | | |
| 2 Macro expansions | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| **C and C++** | | | |
| 1 Null statement (e.g., “;” by itself to indicate an empty body) | | | |
| 2 Expression statements (expressions terminated by semicolons) | | | |
| 3 Expressions separated by semicolons, as in a "for" statement | | | |
| 4 Block statements (e.g., {…} with no terminating semicolon) | | | |
| 5 “{”, “}”, or “};” on a line by itself when part of a declaration | | | |
| 6 “{” or “}” on line by itself when part of an executable statement | | | |
| 7 Conditionally compiled statements (#if, #ifdef, #ifndef) | | | |
| 8 Preprocessor statements other than #if, #ifdef, and #ifndef | | | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |

Figure 3-2 Definition Checklist for Source Statement Counts, Page 3

| Definition name: ______________________ | | Includes | Excludes |
|---|---|---|---|
| **CMS-2** **Listed elements are assigned to** | | | |
| 1 Keywords like SYS-PROC and SYS-DD **statement type –>** | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **COBOL** | | | |
| 1 “PROCEDURE DIVISION”, “END DECLARATIVES”, etc. | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **FORTRAN** | | | |
| 1 END statements | | | |
| 2 Format statements | | | |
| 3 Entry statements | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| **JOVIAL** | | | |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| **Pascal** | | | |
| 1 Executable statements not terminated by semicolons | | | |
| 2 Keywords like INTERFACE and IMPLEMENTATION | | | |
| 3 FORWARD declarations | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |

Figure 3-2 Definition Checklist for Source Statement Counts, Page 4

| Definition name: ______________________________ ______________________________ | | Includes | Excludes |
|---|---|---|---|
| **Listed elements are assigned to statement type –>** | | ▨ | ▨ |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |

**Summary of Statement Types**

**Executable statements**

Executable statements cause runtime actions. They may be simple statements such as assignments, goto's, procedure calls, macro calls, returns, breaks, exits, stops, continues, nulls, no-ops, empty statements, and FORTRAN's END. Or they may be structured or compound statements, such as conditional statements, repetitive statements, and "with" statements. Languages like Ada, C, C++, and Pascal have block statements [begin…end and {…}] that are classified as executable when used where other executable statements would be permitted. C and C++ define expressions as executable statements when they terminate with a semicolon, and C++ has a <declaration> statement that is executable.

**Declarations**

Declarations are nonexecutable program elements that affect an assembler's or compiler's interpretation of other program elements. They are used to name, define, and initialize; to specify internal and external interfaces; to assign ranges for bounds checking; and to identify and bound modules and sections of code. Examples include declarations of names, numbers, constants, objects, types, subtypes, programs, subprograms, tasks, exceptions, packages, generics, macros, and deferred constants. Declarations also include renaming declarations, use clauses, and declarations that instantiate generics. Mandatory begin…end and {…} symbols that delimit bodies of programs and subprograms are integral parts of program and subprogram declarations. Language superstructure elements that establish boundaries for different sections of source code are also declarations. Examples include terms such as PROCEDURE DIVISION, DATA DIVISION, DECLARATIVES, END DECLARATIVES, INTERFACE, IMPLEMENTATION, SYS-PROC, and SYS-DD. Declarations, in general, are never required by language specifications to initiate runtime actions, although some languages permit compilers to implement them that way.

**Compiler Directives**

Compiler directives instruct compilers, preprocessors, or translators (but not runtime systems) to perform special actions. Some, such as Ada's pragma and COBOL's COPY, REPLACE, and USE, are integral parts of the source language. In other languages like C and C++, special symbols like # are used along with standardized keywords to direct preprocessor or compiler actions. Still other languages rely on nonstandardized methods supplied by compiler vendors. In these languages, directives are often designated by special symbols such as #, $, and {$}.

Figure 3-2 Definition Checklist for Source Statement Counts, Page 5

## 3.2. Modifying the Checklist to Meet Special Needs

We have designed the checklist so that when source statements are counted, each statement (or physical line) has exactly one value per attribute. For this to happen, values within an attribute must be both mutually exclusive and collectively exhaustive.

When an organization finds situations where no attribute value exactly fits its measurement needs, there is always a safety net. Users have only to define a new value that does fit their needs and add it to the checklist. One caution: new values added to an attribute may overlap one or more values that already exist. When this happens, users should make clear whether the overlaps are to stand, so that a source statement can take on more than one value for that attribute, or whether the new value is to be treated as a unique subclass of statements that is to be removed from the class or classes of which it was previously a part. If the latter choice is made, which is our recommendation, then the names for all affected values of the attribute should be reworded to describe their new, reduced coverages more accurately.

## 3.3. Assigning Rules for Counting and Classifying Statement Types

When statement counts are made or when automated counters are designed, questions often arise as to whether certain specific language constructs should be included in statement counts. For example, should keywords like *begin* and *end* be counted when they appear on lines by themselves? What about the analogous left and right braces used in C and C++, or seemingly empty statements like nulls, continues, and no-ops?

In constructing our checklist, we found it useful to include sections that would help ensure that some of these less obvious but potentially confusing details associated with classifying and counting statement types get addressed. Opinions with respect to these issues have been known to differ, with dramatic effects on measurement results. (In Chapter 6, we give two examples where large differences of opinion easily occur.) One of the sections we have included applies to many programming languages. This part, which we call **Clarifications (general)**, is shown at the top of page 3 of the checklist. It is followed by further sections that address the special characteristics of some of the more common programming languages. Chapters 4 and 5 include discussions and illustrations of the capabilities that these sections provide.

# 4. Size Attributes and Values: Issues to Consider When Creating Definitions

In this chapter, we define and illustrate the attributes and values used in the definition checklist, and we discuss why the issues they seek to resolve are important. We also provide guidelines and examples for interpreting and using the individual checklist elements.

Our discussions follow the order in which the topics appear in the checklist. The sequence is:

- Statement type
- How produced
- Origin
- Usage
- Delivery
- Functionality
- Replications
- Development status
- Language
- Clarifications (general)
- Clarifications for specific languages

There is a section for each attribute. Each section begins with the purpose of the attribute and a picture of the portion of the checklist that the section addresses. Then, in checklist order, we discuss the values that the attribute can take on. These discussions explain why different values may be important to different users. They also point out some of the implementation issues you may face when including the value or excluding it from your definition or when collecting detailed measurement results.

Before using the checklist to construct definitions for counting source statements, you should read this chapter thoroughly. Understanding the issues the attributes and values address is essential to implementing useful and mutually understood measurement practices.

## 4.1. Statement Type

The statement type attribute classifies source statements and lines according to the principal functions that they perform. The types in the checklist are those that have historically been of interest to project managers, cost modelers, and cost estimators. There are five types: executable statements, declarations, compiler directives, comments, and blank lines. Comments, in turn, have four subtypes: comments on their own lines, comments on lines with other code, banners and nonblank spacers, and blank comments.

| Statement type | Definition ☐ | Data array ☐ | | Includes | Excludes |
|---|---|---|---|---|---|
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | | ▨ | ▨ |
| 1 Executable | | Order of precedence -> | 1 | | |
| 2 Nonexecutable | | | ▨ | ▨ | ▨ |
| 3 Declarations | | | 2 | | |
| 4 Compiler directives | | | 3 | | |
| 5 Comments | | | ▨ | ▨ | ▨ |
| 6 On their own lines | | | 4 | | |
| 7 On lines with source code | | | 5 | | |
| 8 Banners and nonblank spacers | | | 6 | | |
| 9 Blank (empty) comments | | | 7 | | |
| 10 Blank lines | | | 8 | | |
| 11 | | | | | |
| 12 | | | | | |

Figure 4-1 The Statement Type Attribute

### 4.1.1. Executable statements

Executable statements cause runtime actions. They may be simple statements such as assignments, goto's, procedure calls, macro calls, returns, breaks, exits, stops, continues, nulls, no-ops, empty statements, and FORTRAN's *END*. Or they may be structured or compound statements, such as conditional statements, repetitive statements, or *with* statements. Some languages (Ada, C, C++, and Pascal are examples) have block statements (i.e., *begin…end* or *{…}* structures) that they define as executable statements when used where other executable statements would be permitted. Expression-based languages like C and C++ often define expressions to be executable statements when they terminate with a semicolon. C++ even has a *declaration* statement that is executable.

When we count statements, we normally count all these examples as executable statements. We have, however, included provisions in the checklist for you to take a different view. These provisions will be discussed in Sections 4.10 and 4.11 under the topic of **Clarifications**.

In general, executable statements express the operational logic of the program. One characteristic of executable statements is that they can be stepped through with interactive debuggers. Another characteristic is that debuggers can set break points with respect to executable statements, so that computations can be halted to examine intermediate results.

### 4.1.2. Declarations

Declarations are nonexecutable program elements that affect an assembler's or compiler's interpretation of other program elements. They are used to do the following:

- Name, define, and initialize.
- Specify internal and external interfaces.

- Assign ranges for bounds checking.
- Identify and bound modules and sections of code.

Examples include declarations of names, numbers, constants, objects, types, subtypes, programs, subprograms, tasks, exceptions, packages, generics, macros, deferred constants, and the interfaces to any of these software elements. Declarations also include renaming declarations, *use* clauses, and declarations that instantiate generics. Mandatory *begin…end* and *{…}* symbols that delimit bodies of programs or subprograms are not executable statements. Rather, they are integral parts of program and subprogram declarations. Language superstructure elements that establish boundaries for different sections of source code are also declarations. Examples include terms such as *PROCEDURE DIVISION*, *DATA DIVISION*, *DECLARATIVES*, *END DECLARATIVES*, *INTERFACE*, *IMPLEMENTATION*, *SYS-PROC*, and *SYS-DD*. Some languages have separately compiled modules that can be exclusively declarations. Ada's package specifications are examples, and Modula-2 has similar features.

Declarations are never required by language specifications to initiate runtime actions, although some languages permit compilers to implement them that way.

Both the operational and cost distinctions between declarations and executable statements can be fuzzy at times. For example, variables are sometimes initialized statically as part of data declarations and sometimes dynamically at run time. A case in point: Pascal requires that data be initialized with executable statements, but other languages (FORTRAN, C, and Ada, for example) permit equivalent operations to be done at load time through declarations.

There are other ways in which treatments of declarations and executable statements differ across languages. Sometimes these treatments depend on the styles and choices of compiler vendors. For example, when Ada code is compiled, all declarations get elaborated [Ada 83], and elaborations can generate executable code [Ichbiah 86]. In fact, Ada requires that the effects of elaborations must be the same as if they were performed in sequence at run time, although decisions as to how these effects are to be achieved are left to the discretion of individual compiler developers. Static compilation of declarations is permitted, so long as the effects are indistinguishable from runtime elaborations. Therefore, classifying an Ada code element as a declaration does not imply that it is not compiled into some executable machine-level form.

Thus, in the larger perspective, some kinds of declarations may not be very different from executable statements. An unambiguous definition that asks for declarations to be counted separately from executable statements should include explicit resolution of these distinctions, particularly if counts from different languages are to be compared or combined. The clarifications sections on pages 3 through 5 of the checklist are there to help you identify these distinctions and make rules explicit.

One reason for counting declarations separately is that many software professionals attribute different costs to declarations than they do to executable statements. Some popular cost models address this distinction. For example, Boehm's Ada COCOMO model uses a size measure that is the sum of the number of carriage returns in package specifications and the number of semicolons in package bodies [Boehm 89]. PRICE S also treats data declarations differently from executable statements [Park 88]. In both of these cases, separate counts (or estimates) for declarations and executable statements are needed.

Other cost models treat declarations exactly as they treat executable statements, even to the point of referring to them as executable. SLIM is an example [Putnam 91]. In these cases, you can group declarations with executable statements and compiler directives to form a single type for cost estimating purposes, and counting rules can be simplified.

### 4.1.3. Compiler directives

Compiler directives instruct compilers, preprocessors, or translators (but not runtime systems) to perform special actions. Some, such as Ada's pragma and COBOL's *COPY*, *REPLACE,* and *USE*, are integral parts of the source language. In other languages like C and C++, special symbols like # are used along with standardized keywords to direct preprocessor or compiler actions. Still other languages rely on nonstandardized methods supplied by compiler vendors. In these cases, directives are often designated by special symbols such as *#, $,* and *{$}*.

Compiler directives themselves are seldom standardized elements of source languages. Rather, they are special features provided by compiler vendors. They can affect the logical operation of the code. Turning code generation for range checking on or off is just one example.

Compiler directives can also be used for:

- Inserting or expanding copies of other source code at compile time (macro expansions, insertion of include files, and inlining of subroutines and functions are examples).
- Selecting alternative forms of speed and memory optimization.
- Gaining or killing access to operating system features.
- Redefining terms and labels (C's *#define*, for example).
- Controlling the level of diagnostics.
- Defining an object configuration (for example, the size of memory).

Our definition for **Compiler directives** differs from the one in the IEEE draft *Standard for Software Productivity Metrics* [IEEE 92]. The IEEE draft uses **Compiler directive** as a catch-all for statements that fit under no other statement type. We have taken the more specific view that compiler directives are statements that access compiler features that are external to the functions performed by source code languages. If you are uncomfortable with either view, you have two courses of action. You can (1) combine compiler directives with declarations (or with executable statements for that matter) and not worry about the distinctions, or (2) declare and define your own specialized statement types. If you follow either of these paths, you should record your modified definitions on the checklist. We have provided blank lines at the bottom of the statement type block so that you can do this.

### 4.1.4. Comments

Comments are strings and lines of textual characters that have no effect on compiler or program operations. Omitting or revising comments never changes a program's operations or data structures.

Blank comments are physical lines or statements that have comment designators but contain no other visible text or symbols. They are normally the logical equivalent of blank lines. For this reason, most (but not all) people exclude them from counts of source code size.

Many organizations have programming standards that require comments to be placed at the beginning of modules and procedures to report the purpose of the software and to record revision histories. If your organization is interested in measuring and tracking the volume of this work, you may want to create additional comment subtypes to distinguish header and revision history comments from others. The definition checklist has extra spaces in the statement type block for this purpose. Figure 5-14 in Section 5.3.7 shows an example of how a data specification can be constructed for capturing this kind of information.

Comments can come in different forms that serve different purposes. The header and revision history comments just mentioned are but two examples. Whenever you add new classes to the checklist to identify and record special statement types, you should state the explicit rules for identifying the types; otherwise, local interpretations and oversights are almost guaranteed. The supplemental rules forms that we present in Chapter 7 provide a place for recording these rules.

Obtaining counts for comments when counting logical source statements often presents a dilemma that can try the faith of advocates of logical source statement measures. Resolving the dilemma is important for consistent interpretation across different source languages. The dilemma is this:

> Counting comments may not be compatible with counting logical source statements—there may be no simple mechanism for identifying the bounds (beginning and ending) of a logical comment.

For example, in some languages comments that span several lines can arbitrarily have one, two, or many comment designators. Ada is a case in point—every comment starts with two adjacent hyphens and extends only to the end of a line. Most assembly languages have analogous mechanisms for separating comments from code. FORTRAN is similar in that comments start with a *C* in column 1 and extend to the end of a line. In all these cases comments come in short, physical chunks, and the concept of a logical comment statement does not exist. Those who attempt to count logical source statements may have to resort to physical counts to get consistent measures for the volume of comments in their source code.

Pascal, Modula-2, C, and C++, on the other hand, provide more freedom, leading to a wider range of commenting styles. In these languages, comments once started continue until explicitly terminated by an end-of-comment designator. Some organizations enforce standards that require comment terminators at the end of every line. In these cases, counting issues are much as they are in Ada. Other organizations permit comments to continue for several sentences and several lines, terminating only when the comment is signaled to be complete. One comment in these cases may equal several in Ada or FORTRAN. For languages that have commenting features like Pascal, Modula-2, C, or C++, we recommend that summaries of local standards for commenting be attached to reported size measures whenever “logical comments” are counted.

### 4.1.5. Blank lines

Blank lines are physical lines that have no visible textual symbols. They are rarely counted, except perhaps for research studies of the effects of white space on readability, quality, and maintainability.

### 4.1.6. Other statement types

Users who want to add other or more detailed statement types to the checklist may do so. For example, some organizations count the frequencies of different programming constructs (such as *with* and *use* clauses in Ada) to gain insights into programming styles or into progress taking place with respect to learning and adopting new languages and the corresponding effects this has on product designs. This kind of information can often be obtained for little extra cost with commercially available program analyzers.

One word of caution: adding items to the list of statement types in the definition checklist can cause the statement types to overlap. One procedure we recommend to avoid this is to add the new items not to the definition checklist but only to a data specification checklist, and then check the **Data array** box for the statement type attribute. That way overlaps will not corrupt the basic definition of size. Alternatively, you can use the procedure described in Section 5.3.7. Simply add the new items as subclasses of existing statement types and add one or more “other” subclasses so as to complete a mutually exclusive and exhaustive partitioning of the original types. A third option is to add the new items as special classes under the **Clarifications** on pages 3 through 5 of the definition checklist, and then use these classes to construct data specifications.

### 4.1.7. Order of precedence

Some languages permit more than one statement type to appear on a physical line. For these languages, a counting process that classifies statements according to type must have explicit rules for breaking ties. The checklist has small boxes just to the right of the statement types for this purpose. The boxes show the order of precedence that we would use to assign source lines to types when more than one statement type applies. For example, if a comment (precedence level 5) appears on a line with a declaration (precedence level 2), we would classify the line as a declaration. Similarly, lines that contain both declarations and executable statements would be classified as executable statements. Users who wish to use a different precedence order may revise the one in the checklist to meet their needs, so long as they record their changes and make them public.

### 4.1.8. General guidance

Whenever statement types other than blank lines are counted separately or excluded from counts, the rules for counting become dependent on the programming language in which the source code is written. This has direct implications for the construction of automated line and statement counters. When counting rules differ for each source language, either different counters must be prepared or the counters must be designed so that they can parse multiple languages.

Moreover, when counting rules are different for different languages, anyone comparing or combining counts of source statements from different languages must be careful to state their counting rules in ways that make the comparisons or combinations reasonable. Alternatively, they must develop supporting models that account for language differences. In either case, the criteria for comparing or combining counts will depend on the purposes for which the measurements are made. For example, decisions designed to make languages comparable for cost estimators, who prefer counts proportional to effort expended, will be different from those designed to make languages comparable for configuration managers, who prefer measures proportional to physical storage.

Because views on issues like these can legitimately vary, we have included a structure in the checklist for making visible the rules associated with different views. This structure is presented in the **Clarifications** section that begins at the top of page 3 of the checklist and continues on pages 4 and 5. We will discuss this in more detail in Sections 4.10 and 4.11.

## 4.2. How Produced

This attribute identifies the process used to produce individual statements and lines. Classification by production process helps us estimate costs and avoid misleading interpretations when applying size measurements to describe productivity or track progress.

Automated tools that count source lines or statements may not be able to distinguish among lines or statements that are programmed, generated, converted, or copied. Identification of these classes will usually have to be done manually, so that the code can be fed to automated counters in segregated chunks.

| How produced | Definition ☐ | Data array ☐ | Includes | Excludes |
|---|---|---|---|---|
| 1 Programmed | | | | |
| 2 Generated with source code generators | | | | |
| 3 Converted with automated translators | | | | |
| 4 Copied or reused without change | | | | |
| 5 Modified | | | | |
| 6 Removed | | | | |
| 7 | | | | |
| 8 | | | | |

Figure 4-2 The How Produced Attribute

### 4.2.1. Programmed

This class includes all statements prepared by programmers that are not modifications of pre-existing statements. It includes statements that are added when modifying or adapting other bodies of code.

Most people would continue to view programmed statements as programmed after they are processed by automated formatters or pretty printers. In fact, processing source code with formatters and pretty printers is an effective way to simplify and standardize counting rules so as to obtain consistent measurement results.

The use of automated translators can introduce gray areas. But when input code is classified as programmed, output from translators is usually classified as programmed code also, not as generated code. Decisions as to whether to use the input to or the output from translators as the basis for counting will depend on the form in which code is delivered to the customer or the form in which the code will be maintained.

### 4.2.2. Generated with source code generators

Generated source statements are those created by using tools to produce compilable statements automatically from other forms of problem statement or solution description. Examples of source code generators include form generators that translate screen layouts into source code and fourth-generation language translators that produce code in compilable third-generation languages.

Decisions about treating inputs or outputs of source code generators as the source code to be measured often depend on the form in which the code will be maintained.

### 4.2.3. Converted with automated translators

Converted statements are pre-existing statements that are translated automatically or with minor human intervention so that they run on different or newer computers or are expressed in different languages.

There are at least three reasons for counting converted statements separately:

- Converted software usually involves less labor and has lower production costs than programmed or modified code. Sometimes, however, attempts to convert and use existing software can be more expensive than developing new code from scratch. Cost estimators and project trackers will almost always want to address conversions separately from other production classes to avoid erroneous interpretations of productivity and progress.
- Conversion is a form of reuse. Many organizations would like to include counts of converted statements when they compute reuse ratios.
- Converted software has usually had operational use in one or more products prior to being converted. In most cases, the logic associated with the statements has at least been thoroughly tested. As a consequence, converted statements often contain fewer defects than software produced by other means (copying excluded). When you use size measures to compute normalized quality measures such as defect densities, you will normally want to account for the prior testing by tracking your converted code separately.

Since conversions involve bodies of code, not single statements, all code processed into final form by means of conversion (as opposed to copying, modifying, or reprogramming) should be reported in this category, even if some of the statements are unchanged.

### 4.2.4. Copied or reused without change

Copied lines and statements are those taken verbatim from other sources and used as part of the master source code for the new product. This category applies to copies of pre-existing (reused) statements only—copies of master source statements used within the new product are identified and accounted for under the **Replications** attribute as “copies of master statements physically repeated in the master code.”

Since “copied” refers to statements, not modules, the term does not conflict with “removed” or “modified.” If some statements are removed or others are modified, only the remaining unchanged statements are classified as copied. Removed and modified statements are recorded in their own categories.

Moreover, since “copied” refers to statements, not modules, other statements can be inserted before or after individual copied statements, without changing the classes of any of the statements. This implies that copied statements can also be relocated with respect to other statements in a module, without changing their class.

All statements copied from other, already counted statements in the final product are accounted for under the **Replications** attribute.

Size measures for copied software are important for constructing reuse measures and for estimating integration and testing costs. Furthermore, copied and reused code that is delivered in source form must be maintained. This means that measurements of its size are useful for managing and estimating maintenance activities.

### 4.2.5. Modified

Modifications are adaptations made to pre-existing statements so that they can be used in a new product, build, or release. They are common not only during development, but also during maintenance. Each pre-existing statement that we change and retain through an action other than automated translation, we count as modified.

Modified statements exclude removed statements and new statements added as part of a modification process. This means that they can be counted only after software is modified, not before.

Modified statements also exclude relocated statements. For example, if we move seven lines of code from the end of a module to the beginning, we count them as seven lines of copied statements. Since the statements themselves are not modified, this maintains consistency with the statement-based view of attributes on which the checklist is founded.

Some of the other issues you should consider if you plan separate counts for modified statements are consistency with IEEE rules, accounting for different levels of development difficulty, interacting with automated file comparators, and methods for approximating exact counts. We discuss these issues briefly in the paragraphs that follow.

**Consistency with IEEE rules.** The rules we use for counting modified statements are consistent with those in the IEEE draft *Standard for Software Productivity Metrics* [IEEE 92]. Our interpretations of the rules are as follows:

1. If one pre-existing statement is modified to produce one revised statement, then the revised statement that results is counted as a modified statement.
2. If one pre-existing statement is modified to produce a revised statement plus N other statements, then the result is counted as one modified statement and N programmed statements.
3. If P pre-existing statements are modified to produce N new statements and R removed statements, then the number of modified statements M is equal to P–R, and the number of programmed statements is equal to N–M.

**Accounting for different levels of development difficulty.** Since modifications can range from relatively simple changes to extremely complex redesigns, some indication of the extent of prior design, integration, and testing that can be reused may be useful. These evaluations are usually subjective comparisons, made relative to the work required for new code based on new design. If your organization would like to capture this information in a structured way, you may want to consider creating subclasses of modified to record this information.

**Automated counting and file comparators.** If you are counting modified lines but excluding comments, you must be very careful to make your counting rules explicit. Otherwise, confusion can occur when comments are on the same lines as source statements. File comparators will usually classify a line as modified when any change is made on that line, even if the only change is to the comment. Most comparators will give readings as follows:

| Statement modified | Comment modified: Yes | Comment modified: No |
|---|---|---|
| Yes | modified | modified |
| No | **modified** | not modified |

A Comparator's Classification of Modified Lines

This can cause lines to be classified as modified even though no changes have been made to the source statements on them. A sophisticated comparator or specialized counters may be needed to distinguish modified statements from modified comments.

Although file comparators exist that can help distinguish new and modified statements from copied statements, these capabilities are not usually present in today's automated code counters. Until they are, use of comparators in conjunction with counters is likely to be needed to get reliable counts for modified, copied, and programmed statements when the different types are intermixed within individual modules.

**Approximations.** Some organizations attempt to simplify the rules for identifying and counting modified code by applying rules such as this:

```
If a module is changed, then
   if X% or more of the lines are changed, count all lines as new (programmed),
   else count all lines as modified.
```

Values of 25, 30, and 50 are often proposed as appropriate settings for the parameter X.

There are at least two flaws in this logic. First, it presumes you can compute the percentage of lines changed. But to do this, you must have counts for both changed lines and total lines, so that you can form the ratio. If these counts are available, you may as well use them directly rather than substitute a less precise measure that requires additional computation.

The second flaw is that, even when approximations or estimates are substituted for measures of percentage changed, the method fails to deal adequately with statement removal. For example, in a module of 100 lines (or statements) in which 10 are changed and 50 are removed, different people are likely to arrive at entirely different values for the percent changed. The number of opinions would multiply even further if, at the same time, 10 new statements were to be added.

If you want counts of modified statements and if distinguishing modified statements (or lines) from other production classes is difficult or costly, we recommend that you count the total number of statements (or lines) and then estimate the number modified. This method for approximating is at least as effective as the attempts at simplification discussed above. It has the additional benefit of preserving your ability to get consistent counts for generated, converted, copied, and removed statements.

### 4.2.6. Removed

The removed class is used to count all statements that are removed from prior code when that code is copied or modified for use in a new or revised product. Counts for removed code are most useful when planning and managing activities performed to maintain existing software systems. They are also useful in a bookkeeping sense to help ensure that all statements from prior work are accounted for.

Strictly speaking, removed statements do not belong in a size definition because removed code is never part of a product. However, removed is one of the classes accounted for in the IEEE draft *Standard for Software Productivity Metrics* [IEEE 92]. We include it in the checklist for three reasons:

1. Because we want to help users maintain consistency with the IEEE framework.
2. Because measuring removed code is of interest when effort must be expended to locate and identify statements that are to be eliminated.
3. Because counting removed code helps us obtain precise accounting of size changes as programs evolve through successive versions and updates.

A note on terminology: The IEEE draft standard makes a clear distinction between the terms removed and deleted. Specifically, it uses deleted to describe the sum of all statements modified plus all that are removed. Thus:

deleted = modified + removed.

Although this is not a definition we would have chosen, we see no need to argue the point. Because our checklist deals only with mutually exclusive classes, we have no need for the term deleted, and we do not use it in this report. So, in effect, we have adopted the IEEE terminology. Since removed does not overlap any of the other classes under the **How produced** attribute, using it in our checklist presents no problems.

## 4.3. Origin

The **Origin** attribute identifies the prior form (if any) upon which the product software is based. Differences in origins can have substantial impact on development costs, integration costs, and product quality.

Automated source code counters cannot by themselves identify the origins of code submitted to them. If you want separate counts for different origins, you must do one of two things: (1) segregate your statements by origin and feed each origin class to your line or statement counter separately; or (2) tag each module or statement (or the beginnings and endings of each block of statements) with origin codes that the counter can read.

If totals are requested for individual origins, then a separate recording form should be used for each origin so that values for the **Origin** attribute can be identified and assigned when results are entered into databases.

| **Origin** | **Definition** ☐ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|
| 1 | New work: no prior existence | | |
| 2 | Prior work: taken or adapted from | ▨ | ▨ |
| 3 | A previous version, build, or release | | |
| 4 | Commercial, off-the-shelf software (COTS), other than libraries | | |
| 5 | Government furnished software (GFS), other than reuse libraries | | |
| 6 | Another product | | |
| 7 | A vendor-supplied language support library (unmodified) | | |
| 8 | A vendor-supplied operating system or utility (unmodified) | | |
| 9 | A local or modified language support library or operating system | | |
| 10 | Other commercial library | | |
| 11 | A reuse library (software designed for reuse) | | |
| 12 | Other software component or library | | |
| 13 | | | |
| 14 | | | |

Figure 4-3 The Origin Attribute

### 4.3.1. New work: no prior existence

This class includes all statements that implement new designs. Changes to existing designs that implement new features or functions are usually best treated as new designs. The key determinant is the degree of effort (cost) required to design, implement, test, and document the new code.

### 4.3.2. Prior work

This class consists of all statements that are reused from existing software or are produced by adapting or reimplementing existing designs. It includes statements reused or adapted from a previous version of the current product, from other products, and from software developed for other purposes. Examples of prior work include not only operational products, but also reuse libraries, language support libraries, development toolkits, operating system utilities, and pre-existing components, procedures, and code fragments.

**Reasons for identifying prior work.** One reason for identifying statements separately based on prior work is that the total development cost of software produced through reuse is usually less than the cost of new statements based on new design. This information is necessary for estimating and planning, for avoiding overly optimistic views of progress, for correctly interpreting measures of development productivity, and for comparing quality measures across different projects.

A second reason for separately identifying reused elements is that counts of reused statements and designs can be used to track the progress and effectiveness of reuse strategies.  Previous versions of the current product, commercial off-the-shelf software (COTS), government furnished software (GFS), and commercial or reuse libraries are particular examples of prior work that many developers and acquisition agencies will want to track and account for separately, especially if contractor proposals have relied heavily on these elements to reduce development costs.

**Reasons for excluding prior work.**  There are at least two origins that most organizations will want to exclude from source code size measures.  The first is unmodified vendor-supplied language support libraries.  These are normally viewed as part of the source language, almost as extensions of the compiler.  Languages like Ada, C, and C++ routinely use such libraries to provide facilities that other languages imbed in their language definitions.  When these libraries are used without change, they have little impact on development cost.  Most people would view them as part of the environment, not as part of the product.

The second origin that most would exclude from development size measures is unmodified vendor-supplied operating system utilities.  The reasons are similar—these utilities are effectively part of the environment; their existence and use within the product require little testing and documentation; and they add little to development cost.  In maintenance environments, however, the view may differ.  Here changes in utilities can require substantial changes in the programs that use them.

However, if software from either of these classes is modified as part of the development, then the changes will affect integration, testing, and documentation, and the statements in these software units should be counted and included in size measures.  We have listed the element labeled “local or modified language support library or operating system” under the **Origin** attribute expressly for this purpose.

**Methods for excluding prior work.**  Cases can also occur where the arguments for excluding language support and operating system libraries from statement counts would apply to other utility software such as database management, user interface, and statistical packages.  If documentation, integration, and testing of these utilities are not development issues, then you can add additional lines under the **Origin** attribute for the express purpose of excluding these “free” utilities from size counts.

**Relationships between the Origin and How produced attributes.**  Figure 4-4 illustrates the relationships between the **Origin** and **How produced** attributes.  This figure shows the normal relationships that we see during software development.  It is possible, however, to find instances where prior designs are reprogrammed (or regenerated) to produce new code.  In these cases it is appropriate to ascribe prior origins to programmed and generated statements as well.

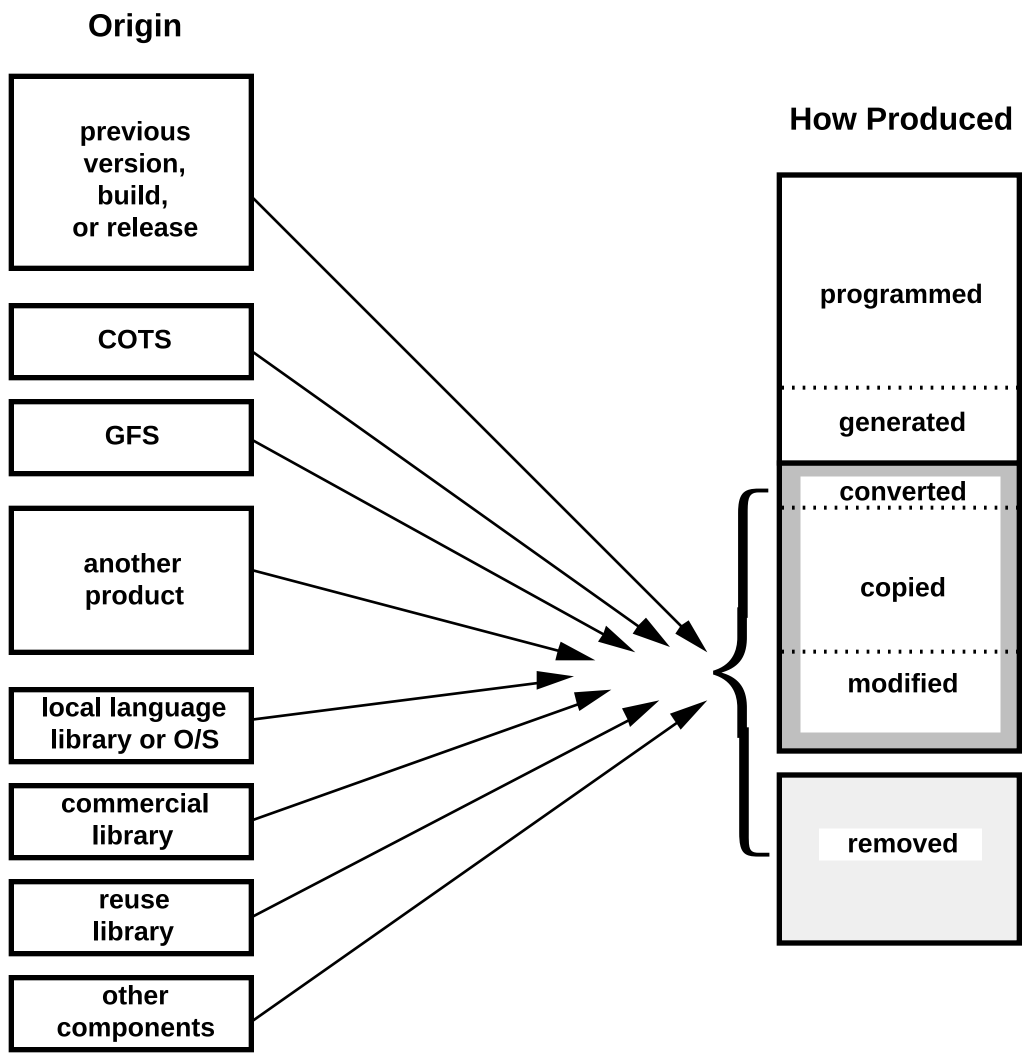


Figure 4-4 Relationships Between Origin and How Produced

## 4.4. Usage

The **Usage** attribute distinguishes between software that is developed or used as an integral part of the primary product and software that is not. Software intended for different purposes is often produced using different design, coding, and testing practices; and these practices can entail different production costs. Identifying these differences is important for estimating costs and schedules and for interpreting reports of productivity, quality, and progress.

| Usage | Definition ☐ | Data array ☐ | Includes | Excludes |
|---|---|---|---|---|
| 1 In or as part of the primary product | | | | |
| 2 External to or in support of the primary product | | | | |
| 3 | | | | |

Figure 4-5 The Usage Attribute

### 4.4.1. In or as part of the primary product

This class comprises all code incorporated into the primary product and all code delivered with or as part of the product that is developed and tested as if it were to operate in the primary product. In addition to code that is actually used, it can include unused (inoperative) source code in delivered source libraries, source code for delivered but unused executable modules or components, and other unused or nondelivered code that is produced with the primary product as the target. (The fact that code is inoperative or not delivered is accounted for elsewhere under other attributes.)

### 4.4.2. External to or in support of the primary product

This class includes all software produced or delivered by the project that is not an integral part of the primary product. Examples include software such as test drivers, tools, simulators, trainers, installation code, diagnostic code, and support utility software whose intended use is outside of the primary product. These kinds of software often have trackable costs associated with them, especially if they are under configuration control. However, they may not be built to the same specification levels or with the same formal controls as primary products. For example, external code and support code may not be subject to the same level of testing as operational software. Consider, for instance, the differences between ground test suites and flight software. In other cases, external or support code may be just a by-product of software development produced under less formal methods. It may also be of a different character or complexity than operational code.

In all these cases, costs and consequences are likely to be different than they are for the primary product. When the differences are significant, the inclusion of external or support code in primary size measures can give misleading indications of progress, productivity, and quality. Similarly, if some reports of size include this code and others do not, the references used for cost estimating become confusing and unreliable. Whenever software that is external to the primary product is of interest, our recommendation is to measure and report this software separately, so that these problems can be avoided.

If external or support code is to be analyzed with cost models, separate accounting is almost always required. In fact, whenever these kinds of software components are to be counted, they are usually best treated as separate products and counted and accounted for separately. In these cases, separate reporting forms should be used to record and report measurement results.

Runtime systems (i.e., lower level systems upon which delivered applications run) are sometimes part of delivered products. Source code for runtime systems may not be delivered or be available to be measured. When it is, it should almost always be measured and reported separately, as it is likely to have different characteristics and costs than application software.

## 4.5. Delivery

The **Delivery** attribute identifies the form and destination of the source statements that are produced. Delivered, in each case, means delivered to the organization that will maintain the software. In commercial organizations, this may mean delivery to the developers themselves.

The four classes are of interest are shown in Figure 4-6. The total number of delivered statements (lines 2 plus 3) reflects the view of the buyer, since delivered statements are what customers perceive they pay for. Configuration control managers, on the other hand, will more likely perceive size as the sum of lines 2, 3, and 5.

| **Delivery** | **Definition** ☐ | **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|---|
| 1 Delivered | | | | |
| 2 Delivered as source | | | | |
| 3 Delivered in compiled or executable form, but not as source | | | | |
| 4 Not delivered | | | | |
| 5 Under configuration control | | | | |
| 6 Not under configuration control | | | | |
| 7 | | | | |

Figure 4-6 The Delivery Attribute

Separate forms should normally be used when recording or reporting each delivery class. Otherwise, attempts to account for this attribute while accounting for others like **Statement type** and **How produced** may overload the inherent two-dimensional nature of sheets of paper.

### 4.5.1. Delivered as source

This class comprises all code that is delivered to the customer in source form, regardless of whether or not compiled versions are also delivered. Separate counts of this class are of interest to maintaining organizations, since delivered source statements are the ones that they will have to support.

### 4.5.2. Delivered in compiled or executable form, but not as source

Statements in this class can occur unintentionally when the developer does not know or loses sight of the fact that software is used or installed. They may also be the result of an oversight—for example, when the developer intended to deliver the source code but forgot. They can even occur intentionally—for example, when language support software supplied by a compiler vendor is included in an executable product but is not delivered as source. There are also instances in contracted developments where a developer's source code is viewed as proprietary and is deliberately withheld.

Whether or not to include counts of the source code for these statements in measures of source code size depends on your perspective:

- From a *cost perspective*, developers will usually need to count source statements for all code delivered in executable form, especially when estimating costs for proposals and when calibrating cost models. This applies particularly when code is developed or changed by the developer, or when it requires integration and testing. Two subclasses that developers will usually not want to count are code drawn without change from (a) language support libraries that have been supplied by the compiler vendor and (b) vendor-supplied operating systems and utilities. Provisions for excluding these statements are addressed under the **Origin** attribute.
- From an *operational perspective*, customers are likely to view statements delivered in executable form (other than operating systems and standard language support libraries) as included in system size, and they will want to count the source statements for this code. Even if delivered only in executable form, the statements do provide functionality that they are paying for.
- From a *maintenance perspective*, the number of statements delivered only in compiled or executable form is rarely of interest, since the class contains no source statements to be maintained. Maintainers of fielded systems have little use for counts of source statements that are not delivered to them. The perspective changes, however, in environments where maintainers are part of the same organization or company that develops the software. Here there may be strong interest in this class, if for no other reason than to track it down so that it does get delivered in source form.

When it comes to practice, developers can count only the source statements they know about and have access to. This rules out all statements available to developers only as executable code. Unfortunately, it also rules out all source statements that developers lose sight of, as well as all statements whose executable forms become included in products without the developers' knowledge. For example, it is not unheard of for entire libraries to be inserted into a product when only a single function is used. (Some issues related to this case are discussed in more detail when we address dead or inoperative code in Section 4.6.)

These difficulties exist in all methods that we know of for counting source statements. If nothing else, recognizing how easy it is to miss source code elements points out how important effective configuration management practices are to the realization of consistently defined measurement results. Those who make or use size measurements should work closely with their configuration managers to ensure that processes are in place that identify and track all source code associated with a project, whether or not the code is obtained from other sources or intentionally used.

### 4.5.3. Not delivered

The statements in this class include all source statements that are produced to support development of the final product but are not delivered to the customer in any form. Nondelivered code is not product code, either in an operational sense or in the view of the customer. Some common examples of nondelivered software include tools, test software, aids to developers, scaffolding, stubs, test drivers, prototypes, simulators, and by-product software.

Nondelivered software is seldom built to the same specification levels or with the same formal controls as the primary product. It is usually produced as just part of the job, and it is often designed and built to be thrown away. For these reasons, counts of nondelivered statements are seldom used as inputs to software cost models. However, tracking of nondelivered software can be useful when working to improve development processes. Software developers may also have reason to measure the size of nondelivered code if they plan to use it elsewhere.

Another reason for excluding counts of nondelivered statements from measures of product size is that developers of cost models have usually excluded nondelivered code when they have constructed and calibrated the equations in their models. Users of these models should use definitions of size that are consistent with the assumptions on which their particular estimating tools are based.

Customers never see nondelivered code, and the factors that describe its costs are almost always different from those of delivered software. When nondelivered code is deemed to be relevant, we recommend that it be counted separately and not included in total size. Here we are more cautious than the IEEE draft *Standard for Software Productivity Metrics* [IEEE 92], which includes nondelivered source statements as part of software size when the statements are developed expressly for the final product.

Some forms of nondelivered software do at times get developed and managed as formal parts of software processes. This can occur, for example, when test cases, test drivers, and simulators get placed under configuration control so that versions and changes can be tracked and supervised. The costs associated with producing and managing these elements may well be of interest. The checklist includes a breakout for this category to assist in distinguishing these elements from delivered software and to provide for measuring their size, should such measures be desired.

## 4.6. Functionality

This attribute identifies whether or not source statements are functional parts of the product. There are two classes: operative code and inoperative code. Inoperative code includes statements that are dead, bypassed, unused, unreferenced, or unaccessed. It also includes dead stores (values computed but never used).

| **Functionality** | **Definition** ☐ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|
| 1 Operative | | | |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | ▨ | ▨ |
| 3 Functional (intentional dead code, reactivated for special purposes) | | | |
| 4 Nonfunctional (unintentionally present) | | | |
| 5 | | | |
| 6 | | | |

Figure 4-7 The Functionality Attribute

### 4.6.1. Operative statements

These statements fulfill the operational requirements of the product. They perform the functions for which the software system was produced. One characteristic of operative statements is that they are or should be included in system testing. For this reason, measures of the size of operative code are the most useful references for tracking progress and for normalizing quality indicators (e.g., defects per thousand lines of code).

### 4.6.2. Inoperative statements (dead, bypassed, unused, unreferenced, or unaccessed)

Inoperative code (dead code, for short) is code that is present in or delivered with a software application but is never referenced, accessed, or used. It produces no runtime results. It can occur in several ways:

- In source modules, either as declarations or format statements that are present but never referenced, or as executable statements that are unreachable because the path to them is blocked or missing.
- In external (source) files such as include files, generics, macros, or interface specifications, as statements that are delivered but never referenced and hence never inserted into compiled modules.
- In compiled modules, as unreachable executable statements that were not identified and eliminated by the compiler.
- In libraries, modules, or objects delivered as part of the product, in the form of source or executable modules, procedures, or methods that are present but never accessed and that the linker has excluded from the load module.

- In the load module, as executable but unreachable procedures, functions, subroutines, or methods.
- In source or executable modules, as the results of computations (dead stores) that are never used.

**Why count dead code?** Dead code can be decidedly different from operational code. It is often not formally designed, not included in system testing, not verified and validated, and not documented. When any of these steps is missing, the quality and development costs of dead code cannot be relied upon as being representative of the rest of the software. In fact, its quality may not even get measured.

Even though we address source-level rather than machine-level size measures in this report, dead code is of interest in at least two cases:

1. When source code gets counted as delivered but is not an operative part of the product.
2. When source code gets omitted from counts even though its inoperative (unreachable), machine-level forms are incorporated in the delivered product.

In the first case, including dead code in size measures may not be in the customer's interest, nor may it be representative of true progress, productivity, or quality. These are all reasons to ensure that steps get taken so that dead code is not accidentally included in measures of system size. In the second case, measuring the amount of dead code that does get into delivered products is a way to track the effectiveness of practices used to manage and eliminate dead code.

**Why counting can be difficult.** Smart compilers and linkers usually remove most dead code from executable products. However, others that are not so smart do not, and some linkers bring in entire libraries when only parts of the libraries are used.

Although smart compilers can detect some kinds of dead code, compilers alone cannot find all of it. Identifying dead code often involves global issues that compilers cannot or do not resolve. One class of examples includes functions, procedures, and methods that are not referenced but that are part of referenced units, modules, and objects. Another class includes unused library routines and functions that get inserted into executable modules along with active library routines and functions. When either of these situations occurs, dead code can exist in large blocks, significantly increasing the size of the load module.

Dead code causes problems for measurement because line counters usually cannot distinguish it from operational code. This presents a serious obstacle not only to separate counting, but also to preventing dead code from contaminating other counts.

**Distinctions between unintentional and functional dead code.** Dead code is usually, but not always, unintentional. Examples of unintentional dead code include loaded but unused routines and functions, unused methods in object-oriented modules, unused declarations, unused computational results, and unreachable sections of source routines and functions. Most customers would not view unintentional dead code as something they pay a developer to develop.  Therefore, it is unlikely that customers would want unintentional dead code included in their primary source code counts.

One example of intentional—and functional—dead code is the use of preset logical switches in conjunction with conditional compilations to bypass statements that are inserted for debugging and testing.  Another example is the use of conditional compilations to bypass statements that are used to reconfigure software to fit alternative physical systems or installations.  Some people prefer to call intentionally inoperative code of these sorts dormant, rather than dead.  Organizations that want to subdivide these forms of intentional dead code can add their specific categories to the checklist in the blank lines that are provided.

**Why be concerned?**  Inoperative code is of concern for several reasons:

- It can inadvertently become operative if it or other code is modified and recompiled, and a path to the code becomes activated.
- It can be expensive, especially in maintenance environments.  Maintenance costs are often a function of the volume of code that maintainers must be conversant with. Maintenance programmers can seldom identify dead code just by looking at a listing—if the code is in the listing, they must spend time analyzing and understanding it.  Even when they know that sections of code are dead, they may be reluctant to excise these sections until they can figure out what the sections were supposed to do and why they have been bypassed.  For these reasons, organizations with maintenance concerns may want to include dead source code in their size counts.
- Dead executable code increases the physical size and memory requirements of executable products, thereby increasing demands on system resources.
- Statement counts that include dead code can corrupt measures of progress, productivity, and quality.  For example, because dead code is seldom tested, including dead code in size measures makes productivities appear higher and defect densities appear lower than they really are.
- A potential exists for grossly misunderstanding size measurements if methods used for counting or omitting dead code are not specified.  For this reason, it is just as important to state the procedures used to exclude dead code from source statement counts as it is to state how dead code is counted.

One of the most disturbing issues about dead code is that, as of today, we do not know how big a problem it presents with respect either to measures of source code size or to the integrity of software products.  The important message is that until we begin to measure dead code, we never will know.

Dead code is thus a management issue, not just a measurement or definition issue.  The fact that dead code is both difficult to avoid and difficult to identify and count separately does not mean that the issue can be ignored.  Rather, it means that users of measures of source code size must state what it takes to make their measurements useful, and then they must work with developers and managers to help make collecting this information possible.  When we apply the checklist to construct example definitions, as we will in Chapter 5, our general recommendation will be to exclude all nonfunctional dead code from measures of software size, but to count and report it separately, if possible.

One action that some organizations have proposed to help manage and account for inoperative code is to treat its identification and removal as a separate element in the work breakdown or cost account structure. Counts of the statements identified and removed can be useful for quantifying this work and for tracking trends over time.

Because inoperative code is so easy to overlook, measurement results that purport to exclude it should be accompanied by descriptions of the procedures used to avoid inadvertently including inoperative code in source code counts.

## 4.7. Replications

This attribute is used to distinguish a product's master source statements from various kinds of copies (replicates) of those statements. Since different kinds of replicates have different costs, this information is important for estimating and planning as well as for measuring and interpreting progress and productivity.

### 4.7.1. Master source statements (originals)

Master source statements are the source statements to which changes are made when revisions are needed. Master source statements may be produced or incorporated into the product by any of the production processes except removal.

| **Replications** | **Definition** ☐ | **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|---|
| 1 Master source statements (originals) | | | | |
| 2 Physical replicates of master statements, stored in the master code | | | | |
| 3 Copies inserted, instantiated, or expanded when compiling or linking | | | | |
| 4 Postproduction replicates—as in distributed, redundant, or reparameterized systems | | | | |
| | | | | |
| 5 | | | | |

Figure 4-8 The Replications Attribute

The source code for macros and generics should be counted as master source statements. All expanded or instantiated reuses of these statements are classified as part of the third replication class (line 3)—as copies inserted, instantiated, or expanded when compiling or linking. Source statements in include files and other blocks that get reused through either manual or automatic copying receive the same treatment—the master code is counted here, the copies and insertions in classes below.

To be counted as delivered source statements, the master source code for macros, generics, include files, and copied blocks must be delivered to the customer. Otherwise, we would view the master code for these statements as being delivered only in executable form.

### 4.7.2. Physical replicates of master statements, stored in the master code

Physical replicates are copies of master source statements that are reproduced—usually in blocks, before compiling—and included in the product through physical storage in master source files. Physical replicates are essentially products of cut-and-paste operations. Because they are produced by physical copying and are stored integrally with master source code, they will usually be counted along with master source statements as part of delivered products. Most organizations will want to do this anyway because, once copied, physical replicates have to be maintained.

Physical replicates are seldom present when include file capabilities are available and used. The use of include files is almost always preferred over cut-and-paste copying, since these files provide central points for making changes that affect all replicates.

Physical replication is a form of reuse. Some organizations may want to include them in their reuse counts. If you would like to do this, you will probably have to mark every physical replicate with a distinctive tag. Otherwise, you will not be able to distinguish physical replicates from master source statements.

Physical replicates do not include backup files. Backup files, if counted at all, are always counted separately.

### 4.7.3. Copies inserted, instantiated, or expanded when compiling or linking

Examples of this class include the results of insertions from include files, the expansions of macros and inlined functions, and the code produced through instantiations of generics. In all cases, the original source statements for these copies are counted as master source statements.

Because statements in this class are never physically present in master source files, they are seldom identified separately as items for configuration control. The master source statements, though, should be deliverable and under configuration control.

The significance of replications in this class, and decisions as to whether or not to count the code they produce or include these counts in size measures, will depend on how and where size measures are used. There are at least four reasons not to count these replications:

- Because automatic reproduction and insertion costs are very low, these statements are of little interest to cost estimators, especially since the design and programming effort to instantiate or insert them is accounted for in the declarations and activation statements that are part of the master source code.

- Because creation is automatic, these statements are usually of little significance to project tracking.
- Because these copies are never physically stored in source form, they are never, by themselves, under configuration control. Hence, they are of little interest to configuration managers.
- Because these copies are identical to statements in the master source code, it would be redundant and confusing to include them in size measures used to normalize most measures of process and product quality.

There are, however, good reasons for counting automatic replications. For example, insertion and instantiation are forms of reuse. Counts of these copies may help to describe product size as customers and reuse managers view it, especially when quantifying the extent to which reuse strategies have been used to reduce development and maintenance costs. Attention must be paid to consistency with other rules, however, since in many cases these copies are nothing but logical equivalents of procedure or function calls.

Another reason to be interested in this class of replications is that copies reproduced or inserted at compile or link time can, in their executable forms, significantly affect the use of runtime resources. Changes in the efficiency of the source code for these copies can have operational impact in many places.

Subcategories of this replication class may be created and added to the checklist when separate accounting for subcategories is desired.

#### 4.7.4. Postproduction replicates

Postproduction replicates are copies of the product that are generated from the master source code and form part of the extended operational system or part of alternative products. They can occur in the following kinds of software:

- High-reliability systems, where multiple copies operate in parallel and comparisons of operational results are used to identify and correct runtime errors.
- Distributed systems, where individual executable copies are installed at multiple nodes.
- Parameterized systems, where recompilations are used to create operational versions that are configured to fit different systems, missions, sites, hardware configurations, or memory models.

There are several reasons for measuring the size of postproduction replicates. For example, postproduction replicates often require independent testing and installation. In these cases, size measurements will be of interest to estimators, planners, and progress trackers. In other instances, postproduction replicates may be the result of conscientious efforts to take advantage of code reuse. Here, size measurements may be wanted to help track and report on reuse trends.

It is best to use separate forms for reporting counts for postproduction replicates. Otherwise, descriptions of system size can become confusing and ambiguous.

## 4.8. Development Status

This attribute describes the degree to which statements under development have progressed toward becoming integral parts of operational products. It is of primary interest during development and maintenance activities. In other situations, measures of size usually refer to a product that has completed system testing.

| **Development status** **Definition** ☐ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|
| *Each statement has one and only one status, usually that of its parent unit.* | ▨ | ▨ |
| 1 Estimated or planned | | |
| 2 Designed | | |
| 3 Coded | | |
| 4 Unit tests completed | | |
| 5 Integrated into components | | |
| 6 Test readiness review completed | | |
| 7 Software (CSCI) tests completed | | |
| 8 System tests completed | | |
| 9 | | |
| 10 | | |
| 11 | | |

Figure 4-9 The Development Status Attribute

When this attribute is used to track development progress, the statements within a software unit will usually all have the same status. This status should be noted on the recording form or otherwise entered into the database.

When the checklist is used to define tracking measures, you should modify or augment the list of states to ensure that it adequately describes the processes you employ. For example, organizations using variations of Boehm’s spiral model for software development [Boehm 88] may want to insert states for tracking the amounts of prototype code they develop.

Once prototyping is complete and final development has started, states are normally listed in a progress-ordered sequence, much like the one shown in Figure 4-9. Here totals for individual states can readily be aggregated and summarized for project tracking and cost analysis. When this is done, it is usually most useful if status information is displayed in cumulative form as the sums of statements that have attained or passed through individual stages (Figure D-1 in Appendix D is an example). The raw data, however, should always be collected and recorded under the identical rules that apply to other attributes—that is, each statement gets assigned one and only one status value.

In many respects, the **Development status** attribute is the most important one on the checklist. As software proceeds through development, work on different components and units will start at different times and proceed at different rates. Individual and cumulative counts for statements that have completed different stages can be extremely helpful in monitoring progress and in adjusting plans and schedules for downstream activities.

For example, since coding precedes unit testing and unit testing precedes integration into components, measures for the number of statements already coded can be used to adjust plans and schedules for unit testing and integration. Also, since counts of the total number of statements coded are available well before integration is complete, they can be used as a baseline for monitoring the degree of progress toward completion of a final product.

Counts of source statements for stages that precede coding are usually estimates. This fact and the way such estimates are used suggest several things:

- The checklist can be used to define the meaning of estimates as well as of actual measurement results.
- The same definition should be used for estimates as for actual measurements.
- Databases and recording forms must distinguish between estimated and measured sizes.
- Estimates should be updated whenever downstream measures are updated, or else the bases for projecting downstream sizes and for interpreting progress toward completion of downstream activities become untrustworthy.

Tracking development status is essential to successful software management. It is one of the few ways in which early warnings of pending overruns and schedule slips can be obtained. We cannot imagine a definition of software size that would not contain provisions for tracking progress during development operations.

## 4.9. Language

This attribute identifies the source languages or languages that are included in a statement count.

Comparing counts of physical or logical source statements across different source languages is difficult. Some professionals argue that language comparisons should be attempted only with a comprehensive model that accounts for language differences. Among their reasons is the fact that the logic content per source statement is not constant (e.g., Ada vs. C, Lisp vs. FORTRAN, 3GL vs. 4GL, assembly vs. 3GL, and microcode vs. anything else). Moreover, some languages make heavy use of support libraries to provide capabilities that are integral to other languages. In these cases, distinguishing between language features and countable code can be difficult. Examples where language support libraries are often treated as part of the language include C, C++, Ada, and Modula-2. Whether to count the code drawn from these libraries as part of a product is debatable, and the decision depends on the purposes for which counts of source statement will be used.

| Language | Definition ☐ Data array ☐ | Includes | Excludes |
|---|---|---|---|
| *List each source language on a separate line.* | | | |
| 1 | | | |
| 2 Job control languages | | | |
| 3 | | | |
| 4 Assembly languages | | | |
| 5 | | | |
| 6 Third generation languages | | | |
| 7 | | | |
| 8 Fourth generation languages | | | |
| 9 | | | |
| 10 Microcode | | | |
| 11 | | | |

Figure 4-10 The Language Attribute

Different languages may also have different costs per logical source statement or physical line. For these reasons as well as those described above, combining counts from different source languages rarely makes sense. When costs or logic content per logical statement or line are different, combining counts from different languages can easily give misleading and erroneous pictures of development progress.

## 4.10. Clarifications (general)

| Clarifications (general) | Listed elements are assigned to statement type –> | Includes | Excludes |
|---|---|---|---|
| 1 Nulls, continues, and no-ops | | | |
| 2 Empty statements (e.g., “;;” and lone semicolons on separate lines) | | | |
| 3 Statements that instantiate generics | | | |
| 4 Begin…end and {…} pairs used as executable statements | | | |
| 5 Begin…end and {…} pairs that delimit (sub)program bodies | | | |
| 6 Logical expressions used as test conditions | | | |
| 7 Expression evaluations used as subprogram arguments | | | |
| 8 End symbols that terminate executable statements | | | |
| 9 End symbols that terminate declarations or (sub)program bodies | | | |
| 10 Then, else, and otherwise symbols | | | |
| 11 Elseif statements | | | |
| 12 Keywords like procedure division, interface, and implementation | | | |
| 13 Labels (branching destinations) on lines by themselves | | | |
| 14 | | | |
| 15 | | | |
| 16 | | | |

Figure 4-11 Clarifications—Inclusion and Statement Typing Rules for Program Elements

This section of the checklist helps us clarify the rules we use to assign statements and lines to the **Statement types** described in Section 4.1. It deals with issues that are common to many programming languages. Subsequent sections address the special features of some frequently used programming languages. In each case, boxes have been provided to the left of the inclusion column so that the program elements listed can be assigned to specific statements types.

### 4.10.1. Nulls, continues, and no-ops

These are statements that, by themselves, have no effect on a program's logic other than to pass to the next statement. They do, however, have uses. For example, they can be used to show that a statement has been deliberately omitted, or they can be placeholders for a future statement. They can also be used as points to attach a label, so that the labeled statement may become a destination or termination of a logic path.

In some cases (e.g., assembly languages), no-op statements can consume execution time. Programmers have been known to use sequences of no-ops to introduce delays for timing purposes. Most organizations now discourage this practice, as it leaves latent problems in the code. Counting no-ops can be a first step toward eliminating these problems. (Source code optimizers effectively prohibit using this trick in higher order languages, even if the language standard does not otherwise rule it out.)

Because null statements, continue statements, and no-ops invoke action (passing to the next statement), most language standards define them to be executable, and most people count them accordingly. This, in fact, is our normal practice. By using line 1 to describe our practices, we avoid confusion when communicating with others who might consider a statement that performs no action to be nonexecutable.

### 4.10.2. Empty statements

Empty statements occur when two or more statement delimiters appear in succession. They can also occur when pairs of delimiters are separated by one or more blank characters or when the delimiters are on different source lines. Moreover, when statement delimiters are separators rather than terminators (as in Pascal), empty statements occur when optional separators are used where they are not required.

Since empty statements have no information content and cause no program action, some people view them as the logical equivalent of blank physical lines. People having this view may want to exclude empty statements from source code counts, especially if they are counting logical source statements. Other people view empty statements as placeholders whose purpose is to call attention to the fact that something has been deliberately omitted or deferred. These people will usually want to include empty statements in source code counts, especially if they are counting physical lines.

Note that, except as implied by location, it is impossible to distinguish an empty executable statement from an empty declaration or from an empty anything else. This reason alone is sufficient to require explicit treatment of empty statements when constructing a definition.

#### 4.10.3. Statements that instantiate generics

Statements that instantiate generics are normally defined by language standards to be declarations. Once created, however, instantiations are accessed via executable statements.

#### 4.10.4. Begin…end and {…} pairs used as executable statements

When these block statements appear in an executable context, they are almost always executable statements, even when empty [Ada 83, Kernighan 88, Stroustrup 87]. They instruct the computer to execute the sequence of statements that is enclosed within the block. Note that this implies that structures like the following count as n+1 logical statements or n+2 physical statements, not as n statements.

```
begin
    statement_1;
    statement_2;
        •
        •
        •
    statement_n;
end;
```

#### 4.10.5. Begin…end and {…} pairs that delimit (sub)program bodies

When these symbols accompany program or subprogram declarations, as some language standards require, they are usually integral parts of the declarations themselves. No language standard that we know of classifies them as executable statements, let alone as logical statements in their own right.

For example, this Pascal program counts as a single declaration “statement”:

```
program ShortExample (input, output, DataFile);
    {declarations go here}
begin
    {executable code goes here}
end.
```

We put quotes around the word “statement” here because, strictly speaking, declarations are not statements. Pascal, like Ada, C, and C++, uses the term “statement” to designate executable statements only. Our use of the term in this report is somewhat more relaxed.

Distinctions between these blocking symbols and those that identify executable block statements require relatively sophisticated source code counters. Users of definitions that make distinctions between executable and nonexecutable statements should be sure to indicate, via a modified definition checklist, when the tools they use are unable to accommodate the definitions they have agreed to employ.

### 4.10.6. Logical expressions used as test conditions

Logical expressions return values. In this sense, they are executable. Moreover, they are usually recognized as executable entities by source code debuggers (single stepping and examination of values is permitted).

For example, a physical line like

> **if** A **then** B;

is viewed by most languages as two logical statements:

> "**if** A **then** < >;" and
> "B".

Here "A" is an expression that returns a value of either *true* or *false*, and the computation of this value is a step that interactive debuggers can count and can pause to examine.

Nevertheless, except possibly for expression-based languages like C and C++ which have facilities for expression statements, entities like "A" are technically parts of other executable statements (the *if* statement, in this case). Because language standards and reference manuals do not usually classify logical expressions as statements in their own right, they are not normally included in counts of logical source statements. However, organizations can certainly have different opinions and use their own specialized counting rules. We include this element on the checklist so that the rules that are used can be made explicit.

### 4.10.7. Expression evaluations used as subprogram arguments

The observations here are essentially the same as those for logical expressions above.

### 4.10.8. End symbols that terminate executable statements

These end symbols are not normally logical statements by themselves—they are merely parts of the logical statements they terminate, even when on separate lines.

### 4.10.9. End symbols that terminate declarations or (sub)program bodies

Again, these are not normally statements by themselves—they are merely parts of the statements or declarations they terminate, even when on separate lines. FORTRAN's *END* statement is an exception. It is defined by the language standard to be executable.

### 4.10.10. Then, else, and otherwise symbols

These symbols are normally parts of statements, not statements by themselves. They may, however, bracket or delimit other (usually executable) statements. If they do, the rules for recognizing these delimiters should be spelled out on the supplemental rules form, so that there are no misunderstandings when designing or using automated source code counters.

### 4.10.11. Elseif and elsif statements

Different languages classify these differently. For example, Ada treats its *elsif* as an integral part of the *if* statement in which it is used. This makes it consistent with the way most languages treat case statements. In other languages, *elsif* or *elseif* statements can be executable statements in their own right, each having its own conditional clause and executable part, although the executable part may be empty. Statements within an executable part are counted as executable statements in their own right, just as are the contents of *begin…end* pairs.

### 4.10.12. Keywords like procedure division, interface, and implementation

Keywords and expressions like these are sometimes used by language standards to identify and bound specific sections of code. Since they are not executable, they are declarations. They declare to the compiler and to the reader the kind of code block that they initiate or terminate.

### 4.10.13 Labels (branching destinations) on lines by themselves

A test and branch to the end of a block is often used to bypass the remaining statements in a block or to exit a loop. Although many languages require this branch to be made to an executable statement, others do not. Some languages do not even require that a statement be present—a label alone may suffice. When this occurs, we need to know the rules used to determine: (a) do we count the label? and (b) if we are classifying statements by type, how do we classify it?

## 4.11. Clarifications for Specific Languages

The remaining sections of the checklist provide places for describing the special counting rules used with different programming languages. These sections are far from complete, either in terms of languages covered or details addressed. We strongly encourage you to add to the examples (and languages) that we list. To illustrate the kinds of points that need to be settled, we draw your attention to three often disputed cases: C's stylistic use of braces, the treatment of expression statements in expression-based languages, and FORTRAN's *FORMAT* statement.

### 4.11.1. Stylistic use of braces

The C and C++ languages use braces to identify and bound blocks of code. It is customary for programmers to place these braces on lines by themselves. The purpose is to guide the eye when reading source code, so that beginnings and endings of sections and block statements can be quickly identified.

This stylistic use of braces raises the question, "Should solitary braces be included in counts of physical source lines?" The answers are almost guaranteed to generate heated debate. On the one hand, a single brace bears little semblance to our usual view of a source statement. Displaying one on a line by itself doesn't do anything to a program's logic. For this reason, many cost analysts advocate treating solitary braces as blank lines, arguing that they serve the same purpose that white space does in the coding styles associated with other languages.

On the other hand, users of other languages frequently count lines that perform equivalent functions. For example, several languages have *begin* and *end* keywords that become used in exactly the same stylistic way as C and C++ use braces. Organizations that use these languages often include these keywords in their line counts, even when they are on lines by themselves.

In either case, you should make your counting rules explicit if you want to know what your source line counts represent. Moreover, if you want to compare measurement results across different languages, you should probably make the rules the same for all languages. Since we have no statistical evidence to support a preference for exclusion over inclusion, we are inclined (for the time being, at least) to include these lines in our own counts. They are, after all, physical lines of code. This simplifies the counting process because it means that no special rules need be developed for skipping these lines. It also avoids introducing what is essentially a logical source statement concept into a physical line count. If we want to exclude solitary braces and solitary keywords, we can always elect to count logical source statements instead of physical lines.

### 4.11.2. Expression statements

C and C++ are expression-based languages. In these languages, expressions such as *x = 0* or *i*++ become statements when followed by a semicolon. Expression statements are executed by evaluating the expression and then discarding the result. This is useful if evaluation involves a side effect, such as assigning a value to a variable or performing input or output.

The lines that follow provide four examples. In each case, the line would be an expression if it did not have a semicolon. The presence of the semicolon converts the expression into a statement.

```
distance = speed * time;      /* assigns the product to distance */
printf("Hello world!");       /* calls the function printf */
++counter;                    /* adds 1 to counter */
(x < y) ? ++x : ++y;          /* increments x when x is
                                   smaller than y, else increments y */
```

Note that the first two lines illustrate an assignment and a function call, actions that almost all languages treat as executable statements. The third and fourth lines have executable counterparts in many assembly languages.

If you are counting logical source statements and if your practice is to classify expression statements as executable statements, you must be both explicit and complete when stating your rules with respect to semicolons. For example, expressions followed by semicolons occur also as test conditions in *for* loops. The rules you state should make it clear whether or not you count these constructs as individual statements.

### 4.11.3. Format statements

Counting rules for format statements vary considerably among different cost estimators. To help you decide how you wish to count format statements, we offer the following brief discussion.

Format statements are statements that provide (often static) formatting rules and textual labeling for information that is displayed on screens, printed on output reports, or sent to files or other output devices. They are also used to define the ways in which input information is interpreted and edited. In FORTRAN, format statements direct "the editing between the internal representation and the character strings of a record or a sequence of records in a file" [FORTRAN 77]. The FORTRAN language standard classifies format statements as nonexecutable and treats them as constants or data to be interpreted and processed at run time. Thus, in the terminology of the checklist, they are declarations.

Some cost models adopt this view, treating format statements as data and excluding them from counts of executable statements (PRICE S is an example). These models account for the effort involved in designing, coding, and testing format statements through the weights they give to the associated executable write and print statements. Users of these cost models will usually want to measure and account for format statements individually, so that they can adjust their size measures to fit the models they use.

Because format statements are sometimes treated as data and at other times as declarations or executable statements, size definitions that combine counts of format statements with counts for other statement types should make the rules visible in the respective language-specific clarifications. For languages without specially designated format statements, this may not be an issue. Even here, however, counting the equivalent operations—and counting them equivalently—may be important when comparing size measures from different languages. If so, then the ground rules for identifying and counting formatting operations should be spelled out.

# 5. Using the Checklist to Define Size Measures—Nine Examples

The definition checklist can be used in two different ways: to describe size measurements after they are made and to prescribe how we want them to be made.

When we introduced the checklist in Chapter 3, our focus was on descriptive use. We now present examples in which we apply the checklist, together with the results of our discussions in Chapter 4, to construct definitions (prescriptions) for two specific measures of software size—physical source lines of code (SLOC) and logical source statements. We then present several examples of data specifications, so that you can see how a common definition can be used by different people while giving each the flexibility to collect the data they need. This flexibility is important because it permits us to hold basic definitions stable while adjusting to the changing information needs that accompany advancing levels of process maturity.

Note that although we present the definitions and data specifications in the context of requirements for inclusion and exclusion, you can also use the checklist simply to record and report the rules and practices associated with measurements that have already been made. Indeed, it can be interesting and informative to do this for size values in the databases you currently use. For example, cost estimators who use historical data for calibrating cost models will find the checklist useful in pinning down and recording the exact meanings of the size data they have in their files, especially if the data has come from different sources or was collected at different points in time. Once estimators have the insight that completed checklists provide, they can adjust their models and calibrations to fit the definitions that were actually used when the measurements were made.

This descriptive approach is essentially the one we used when we constructed the checklist. First we posed the question: “What is it that we need to know when we use the measures of source code size that are reported to us?” We then observed that if we can capture this information in a structured framework, it is easy to turn the framework about and use it in a prescriptive sense to tell others how to collect and record the information we need.

Figures 5-1 and 5-2 show our example definitions for physical source lines and logical source statements. They represent consensus recommendations from those who have helped prepare and review this report. Organizations with needs that these examples do not satisfy should adjust our recommendations to fit their individual requirements.

Our goals in preparing the examples are to define, to the extent possible, measures that can be used comparably across organizations and consistently over time. We believe that as software organizations move to higher process maturities, they should not be forced to discard historical data collected along the way just because their information needs change or become more refined.

A word of caution: Although the checklist can be used to create and report many different definitions of size, this does not imply that having many definitions is a good idea. A consistent definition for size within projects is a prerequisite for project planning and control, and a consistent definition across projects is a prerequisite for process improvement. Moreover, once a consistent definition is in place, organizations will no longer have to resort to reconstructing the definitions used to capture historical size data.

This chapter deals primarily with the coverage of definitions and data specification. The emphasis here is more on what to count than how to count. Additional points that need to be addressed to fully define for a counting unit are discussed in Chapters 6 (Defining the Counting Unit) and 7 (Reporting Other Rules and Practices—The Supplemental Forms).

## 5.1. Physical Source Lines—A Basic Definition

Figure 5-1 shows our recommended definition for counting physical source lines of code. This definition explicitly identifies the values for each attribute that we include in or exclude from our statement counts. The checks in the **Definition** boxes for eight of the attributes distinguish the rules for those attributes from requests for counts of individual data elements. The check in the **Definition** box for the **Language** attribute shows that we want a separate count for each source language we use. In Section 5.3, when we introduce data specifications, we will illustrate other cases where we use the checklist to explain the rules for counting individual data elements.

Our treatment of the **Development status** attribute on page 2 of the checklist may at first glance seem counterintuitive. Here we have excluded statements from everything but the final product. When we build or modify software systems, we will certainly be interested in counts for the numbers of statements in intermediate stages of development. These progress measures, however, are best dealt with via data specifications like those in Section 5.3. In most other circumstances, we want a single definition that we can use for purposes such as estimating the size of a project, tracking convergence to a planned target, describing a product to a customer, and calibrating cost models. The single check in the **Includes** column for development status provides this definition. It means that we define software size to be the size of the finished product.

# Definition Checklist for Source Statement Counts

Definition name: ***Physical Source Lines of Code (basic definition)*** Date: ***8/7/92***

Originator: ***SEI***

| Measurement unit: | |
|---|---|
| Physical source lines | ✓ |
| Logical source statements | |

| Statement type | Definition ✓ Data array ☐ | | Includes | Excludes |
|---|---|---|---|---|
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | ▨ | ▨ |
| 1 Executable | **Order of precedence ->** | 1 | ✓ | |
| 2 Nonexecutable | | ▨ | ▨ | ▨ |
| 3 Declarations | | 2 | ✓ | |
| 4 Compiler directives | | 3 | ✓ | |
| 5 Comments | | ▨ | ▨ | ▨ |
| 6 On their own lines | | 4 | | ✓ |
| 7 On lines with source code | | 5 | | ✓ |
| 8 Banners and nonblank spacers | | 6 | | ✓ |
| 9 Blank (empty) comments | | 7 | | ✓ |
| 10 Blank lines | | 8 | | ✓ |
| 11 | | | | |
| 12 | | | | |

| How produced | Definition ✓ Data array ☐ | Includes | Excludes |
|---|---|---|---|
| 1 Programmed | | ✓ | |
| 2 Generated with source code generators | | ✓ | |
| 3 Converted with automated translators | | ✓ | |
| 4 Copied or reused without change | | ✓ | |
| 5 Modified | | ✓ | |
| 6 Removed | | | ✓ |
| 7 | | | |
| 8 | | | |

| Origin | Definition ✓ Data array ☐ | Includes | Excludes |
|---|---|---|---|
| 1 New work: no prior existence | | ✓ | |
| 2 Prior work: taken or adapted from | | ▨ | ▨ |
| 3 A previous version, build, or release | | ✓ | |
| 4 Commercial, off-the-shelf software (COTS), other than libraries | | ✓ | |
| 5 Government furnished software (GFS), other than reuse libraries | | ✓ | |
| 6 Another product | | ✓ | |
| 7 A vendor-supplied language support library (unmodified) | | | ✓ |
| 8 A vendor-supplied operating system or utility (unmodified) | | | ✓ |
| 9 A local or modified language support library or operating system | | ✓ | |
| 10 Other commercial library | | ✓ | |
| 11 A reuse library (software designed for reuse) | | ✓ | |
| 12 Other software component or library | | ✓ | |
| 13 | | | |
| 14 | | | |

| Usage | Definition ✓ Data array ☐ | Includes | Excludes |
|---|---|---|---|
| 1 In or as part of the primary product | | ✓ | |
| 2 External to or in support of the primary product | | | ✓ |
| 3 | | | |

Figure 5-1 Definition for Physical Source Lines of Code (SLOC)

Definition name: ***Physical Source Lines of Code (basic definition)***

| **Delivery** Definition [✓] Data array [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 Delivered | | |
| 2 Delivered as source | ✓ | |
| 3 Delivered in compiled or executable form, but not as source | ✓ | |
| 4 Not delivered | | |
| 5 Under configuration control | | ✓ |
| 6 Not under configuration control | | ✓ |
| 7 | | |

| **Functionality** Definition [✓] Data array [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 Operative | ✓ | |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | |
| 3 Functional (intentional dead code, reactivated for special purposes) | ✓ | |
| 4 Nonfunctional (unintentionally present) | | ✓ |
| 5 | | |
| 6 | | |

| **Replications** Definition [✓] Data array [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 Master source statements (originals) | ✓ | |
| 2 Physical replicates of master statements, stored in the master code | ✓ | |
| 3 Copies inserted, instantiated, or expanded when compiling or linking | | ✓ |
| 4 Postproduction replicates—as in distributed, redundant, or reparameterized systems | | ✓ |
| 5 | | |

| **Development status** Definition [✓] Data array [ ] | **Includes** | **Excludes** |
|---|---|---|
| *Each statement has one and only one status, usually that of its parent unit.* | | |
| 1 Estimated or planned | | ✓ |
| 2 Designed | | ✓ |
| 3 Coded | | ✓ |
| 4 Unit tests completed | | ✓ |
| 5 Integrated into components | | ✓ |
| 6 Test readiness review completed | | ✓ |
| 7 Software (CSCI) tests completed | | ✓ |
| 8 System tests completed | ✓ | |
| 9 | | |
| 10 | | |
| 11 | | |

| **Language** Definition [ ] Data array [✓] | **Includes** | **Excludes** |
|---|---|---|
| *List each source language on a separate line.* | | |
| 1 ***Separate totals for each language*** | ✓ | |
| 2 Job control languages | | |
| 3 | | |
| 4 Assembly languages | | |
| 5 | | |
| 6 Third generation languages | | |
| 7 | | |
| 8 Fourth generation languages | | |
| 9 | | |
| 10 Microcode | | |
| 11 | | |

Figure 5-1 Definition for Physical Source Lines of Code (SLOC), Page 2

| Definition name: ***Physical Source Lines of Code (basic definition)*** | | **Includes** | **Excludes** |
|---|---|---|---|
| **Clarifications (general)** **Listed elements are assigned to statement type –>** | | | |
| 1 Nulls, continues, and no-ops | *1* | ✓ | |
| 2 Empty statements (e.g., “;;” and lone semicolons on separate lines) | *1* | ✓ | |
| 3 Statements that instantiate generics | *3* | ✓ | |
| 4 Begin…end and {…} pairs used as executable statements | *1* | ✓ | |
| 5 Begin…end and {…} pairs that delimit (sub)program bodies | *3* | ✓ | |
| 6 Logical expressions used as test conditions | *1* | ✓ | |
| 7 Expression evaluations used as subprogram arguments | *1* | ✓ | |
| 8 End symbols that terminate executable statements | *1* | ✓ | |
| 9 End symbols that terminate declarations or (sub)program bodies | *3* | ✓ | |
| 10 Then, else, and otherwise symbols | *1* | ✓ | |
| 11 Elseif statements | *1* | ✓ | |
| 12 Keywords like procedure division, interface, and implementation | *3* | ✓ | |
| 13 Labels (branching destinations) on lines by themselves | *1* | ✓ | |
| 14 | | | |
| 15 | | | |
| 16 | | | |
| **Clarifications (language specific)** | | | |
| **Ada** | | | |
| 1 End symbols that terminate declarations or (sub)program bodies | *3* | ✓ | |
| 2 Block statements (e.g., begin…end) | *1* | ✓ | |
| 3 With and use clauses | *3* | ✓ | |
| 4 When (the keyword preceding executable statements) | *1* | ✓ | |
| 5 Exception (the keyword, used as a frame header) | *3* | ✓ | |
| 6 Pragmas | *4* | ✓ | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **Assembly** | | | |
| 1 Macro calls | *1* | ✓ | |
| 2 Macro expansions | | | ✓ |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| **C and C++** | | | |
| 1 Null statement (e.g., “;” by itself to indicate an empty body) | *1* | ✓ | |
| 2 Expression statements (expressions terminated by semicolons) | *1* | ✓ | |
| 3 Expressions separated by semicolons, as in a "for" statement | *1* | ✓ | |
| 4 Block statements (e.g., {…} with no terminating semicolon) | *1* | ✓ | |
| 5 “{”, “}”, or “};” on a line by itself when part of a declaration | *3* | ✓ | |
| 6 “{” or “}” on line by itself when part of an executable statement | *1* | ✓ | |
| 7 Conditionally compiled statements (#if, #ifdef, #ifndef) | *4* | ✓ | |
| 8 Preprocessor statements other than #if, #ifdef, and #ifndef | *4* | ✓ | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |

Figure 5-1 Definition for Physical Source Lines of Code (SLOC), Page 3

| Definition name: ***Physical Source Lines of Code (basic definition)*** | | Includes | Excludes |
|---|---|---|---|
| **CMS-2** **Listed elements are assigned to** | | | |
| 1 Keywords like SYS-PROC and SYS-DD **statement type –>** | *3* | ✔ | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **COBOL** | | | |
| 1 “PROCEDURE DIVISION”, “END DECLARATIVES”, etc. | *3* | ✔ | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **FORTRAN** | | | |
| 1 END statements | *1* | ✔ | |
| 2 Format statements | *3* | ✔ | |
| 3 Entry statements | *3* | ✔ | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| **JOVIAL** | | | |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| **Pascal** | | | |
| 1 Executable statements not terminated by semicolons | *1* | ✔ | |
| 2 Keywords like INTERFACE and IMPLEMENTATION | *3* | ✔ | |
| 3 FORWARD declarations | *3* | ✔ | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |

Figure 5-1 Definition for Physical Source Lines of Code (SLOC), Page 4

| Definition name: ***Physical Source Lines of Code (basic definition)*** | | **Includes** | **Excludes** |
|---|---|---|---|
| **Listed elements are assigned to statement type –>** | | | |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |

**Summary of Statement Types**

**Executable statements**

Executable statements cause runtime actions. They may be simple statements such as assignments, goto’s, procedure calls, macro calls, returns, breaks, exits, stops, continues, nulls, no-ops, empty statements, and FORTRAN’s END. Or they may be structured or compound statements, such as conditional statements, repetitive statements, and “with” statements. Languages like Ada, C, C++, and Pascal have block statements [begin…end and {…}] that are classified as executable when used where other executable statements would be permitted. C and C++ define expressions as executable statements when they terminate with a semicolon, and C++ has a <declaration> statement that is executable.

**Declarations**

Declarations are nonexecutable program elements that affect an assembler’s or compiler’s interpretation of other program elements. They are used to name, define, and initialize; to specify internal and external interfaces; to assign ranges for bounds checking; and to identify and bound modules and sections of code. Examples include declarations of names, numbers, constants, objects, types, subtypes, programs, subprograms, tasks, exceptions, packages, generics, macros, and deferred constants. Declarations also include renaming declarations, use clauses, and declarations that instantiate generics. Mandatory begin…end and {…} symbols that delimit bodies of programs and subprograms are integral parts of program and subprogram declarations. Language superstructure elements that establish boundaries for different sections of source code are also declarations. Examples include terms such as PROCEDURE DIVISION, DATA DIVISION, DECLARATIVES, END DECLARATIVES, INTERFACE, IMPLEMENTATION, SYS-PROC, and SYS-DD. Declarations, in general, are never required by language specifications to initiate runtime actions, although some languages permit compilers to implement them that way.

**Compiler Directives**

Compiler directives instruct compilers, preprocessors, or translators (but not runtime systems) to perform special actions. Some, such as Ada’s pragma and COBOL’s COPY, REPLACE, and USE, are integral parts of the source language. In other languages like C and C++, special symbols like # are used along with standardized keywords to direct preprocessor or compiler actions. Still other languages rely on nonstandardized methods supplied by compiler vendors. In these languages, directives are often designated by special symbols such as #, $, and {$}.

Figure 5-1 Definition for Physical Source Lines of Code (SLOC), Page 5

## 5.2. Logical Source Statements—A Basic Definition

Figure 5-2 shows our recommendation for a definition for counting logical source statements. This definition is identical to the one shown for physical source lines, except for the **Clarifications** sections on pages 3 and 4 of the checklist. The similarities in treatment of attributes should come as no surprise—the coverage and tracking issues are the same for the two measures. The differences in the **Clarifications** sections address details of counting that have to be spelled out to make measures of logical source statements rigorous and repeatable.

Organizations implementing this definition for logical source statements should review carefully—and augment, where necessary—the language-specific clarifications. They should also complete and append a supplemental rules form like the one in Chapter 7. These steps are even more important here than they are for physical source lines of code. Without complete sets of clarifications and supplemental rules for each language measured, important low-level decisions will be left to the judgment of others. If this happens, then users of measurement results may never know what decisions were implemented, and measurements made by different teams may have entirely different meanings. What's more, organizations will have no reason to believe three or four years later that they are still following their original counting practices.

## Definition Checklist for Source Statement Counts

Definition name: ***Logical Source Statements (basic definition)*** Date: ***8/7/92***

Originator: ***SEI***

| | | | | |
|---|---|---|---|---|
| **Measurement unit:** | **Physical source lines** ☐ **Logical source statements** ✓ | | | |
| **Statement type** | **Definition** ✓ **Data array** ☐ | | **Includes** | **Excludes** |
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | | |
| 1 Executable | **Order of precedence ->** | 1 | ✓ | |
| 2 Nonexecutable | | | | |
| 3 Declarations | | 2 | ✓ | |
| 4 Compiler directives | | 3 | ✓ | |
| 5 Comments | | | | |
| 6 On their own lines | | 4 | | ✓ |
| 7 On lines with source code | | 5 | | ✓ |
| 8 Banners and nonblank spacers | | 6 | | ✓ |
| 9 Blank (empty) comments | | 7 | | ✓ |
| 10 Blank lines | | 8 | | ✓ |
| 11 | | | | |
| 12 | | | | |
| **How produced** | **Definition** ✓ **Data array** ☐ | | **Includes** | **Excludes** |
| 1 Programmed | | | ✓ | |
| 2 Generated with source code generators | | | ✓ | |
| 3 Converted with automated translators | | | ✓ | |
| 4 Copied or reused without change | | | ✓ | |
| 5 Modified | | | ✓ | |
| 6 Removed | | | | ✓ |
| 7 | | | | |
| 8 | | | | |
| **Origin** | **Definition** ✓ **Data array** ☐ | | **Includes** | **Excludes** |
| 1 New work: no prior existence | | | ✓ | |
| 2 Prior work: taken or adapted from | | | | |
| 3 A previous version, build, or release | | | ✓ | |
| 4 Commercial, off-the-shelf software (COTS), other than libraries | | | ✓ | |
| 5 Government furnished software (GFS), other than reuse libraries | | | ✓ | |
| 6 Another product | | | ✓ | |
| 7 A vendor-supplied language support library (unmodified) | | | | ✓ |
| 8 A vendor-supplied operating system or utility (unmodified) | | | | ✓ |
| 9 A local or modified language support library or operating system | | | ✓ | |
| 10 Other commercial library | | | ✓ | |
| 11 A reuse library (software designed for reuse) | | | ✓ | |
| 12 Other software component or library | | | ✓ | |
| 13 | | | | |
| 14 | | | | |
| **Usage** | **Definition** ✓ **Data array** ☐ | | **Includes** | **Excludes** |
| 1 In or as part of the primary product | | | ✓ | |
| 2 External to or in support of the primary product | | | | ✓ |
| 3 | | | | |

Figure 5-2 Definition for Logical Source Statements

Definition name: ***Logical Source Statements***
***(basic definition)***

| Delivery | Definition ✓ | Data array ☐ | Includes | Excludes |
|---|---|---|---|---|
| 1 Delivered | | | | |
| 2 Delivered as source | | | ✓ | |
| 3 Delivered in compiled or executable form, but not as source | | | ✓ | |
| 4 Not delivered | | | | |
| 5 Under configuration control | | | | ✓ |
| 6 Not under configuration control | | | | ✓ |
| 7 | | | | |

| Functionality | Definition ✓ | Data array ☐ | Includes | Excludes |
|---|---|---|---|---|
| 1 Operative | | | ✓ | |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | | | |
| 3 Functional (intentional dead code, reactivated for special purposes) | | | ✓ | |
| 4 Nonfunctional (unintentionally present) | | | | ✓ |
| 5 | | | | |
| 6 | | | | |

| Replications | Definition ✓ | Data array ☐ | Includes | Excludes |
|---|---|---|---|---|
| 1 Master source statements (originals) | | | ✓ | |
| 2 Physical replicates of master statements, stored in the master code | | | ✓ | |
| 3 Copies inserted, instantiated, or expanded when compiling or linking | | | | ✓ |
| 4 Postproduction replicates—as in distributed, redundant, or reparameterized systems | | | | ✓ |
| 5 | | | | |

| Development status | Definition ✓ | Data array ☐ | Includes | Excludes |
|---|---|---|---|---|
| *Each statement has one and only one status, usually that of its parent unit.* | | | | |
| 1 Estimated or planned | | | | ✓ |
| 2 Designed | | | | ✓ |
| 3 Coded | | | | ✓ |
| 4 Unit tests completed | | | | ✓ |
| 5 Integrated into components | | | | ✓ |
| 6 Test readiness review completed | | | | ✓ |
| 7 Software (CSCI) tests completed | | | | ✓ |
| 8 System tests completed | | | ✓ | |
| 9 | | | | |
| 10 | | | | |
| 11 | | | | |

| Language | Definition ☐ | Data array ✓ | Includes | Excludes |
|---|---|---|---|---|
| *List each source language on a separate line.* | | | | |
| 1 | ***Separate totals for each language*** | | ✓ | |
| 2 Job control languages | | | | |
| 3 | | | | |
| 4 Assembly languages | | | | |
| 5 | | | | |
| 6 Third generation languages | | | | |
| 7 | | | | |
| 8 Fourth generation languages | | | | |
| 9 | | | | |
| 10 Microcode | | | | |
| 11 | | | | |

Figure 5-2 Definition for Logical Source Statements, Page 2

| Definition name: ***Logical Source Statements (basic definition)*** | | Includes | Excludes |
|---|---|---|---|
| **Clarifications (general)** **Listed elements are assigned to statement type –>** | | | |
| 1 Nulls, continues, and no-ops | *1* | ✔ | |
| 2 Empty statements (e.g., “;;” and lone semicolons on separate lines) | | | ✔ |
| 3 Statements that instantiate generics | *3* | ✔ | |
| 4 Begin…end and {…} pairs used as executable statements | *1* | ✔ | |
| 5 Begin…end and {…} pairs that delimit (sub)program bodies | | | ✔ |
| 6 Logical expressions used as test conditions | | | ✔ |
| 7 Expression evaluations used as subprogram arguments | | | ✔ |
| 8 End symbols that terminate executable statements | | | ✔ |
| 9 End symbols that terminate declarations or (sub)program bodies | | | ✔ |
| 10 Then, else, and otherwise symbols | | | ✔ |
| 11 Elseif statements | *1* | ✔ | |
| 12 Keywords like procedure division, interface, and implementation | *3* | ✔ | |
| 13 Labels (branching destinations) on lines by themselves | | | ✔ |
| 14 | | | |
| 15 | | | |
| 16 | | | |
| **Clarifications (language specific)** | | | |
| **Ada** | | | |
| 1 End symbols that terminate declarations or (sub)program bodies | | | ✔ |
| 2 Block statements (e.g., begin…end) | *1* | ✔ | |
| 3 With and use clauses | *3* | ✔ | |
| 4 When (the keyword preceding executable statements) | | | ✔ |
| 5 Exception (the keyword, used as a frame header) | *3* | ✔ | |
| 6 Pragmas | *4* | ✔ | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **Assembly** | | | |
| 1 Macro calls | *1* | ✔ | |
| 2 Macro expansions | | | ✔ |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| **C and C++** | | | |
| 1 Null statement (e.g., “;” by itself to indicate an empty body) | | | ✔ |
| 2 Expression statements (expressions terminated by semicolons) | *1* | ✔ | |
| 3 Expressions separated by semicolons, as in a "for" statement | *1* | ✔ | |
| 4 Block statements (e.g., {…} with no terminating semicolon) | *1* | ✔ | |
| 5 “{”, “}”, or “};” on a line by itself when part of a declaration | | | ✔ |
| 6 “{” or “}” on line by itself when part of an executable statement | | | ✔ |
| 7 Conditionally compiled statements (#if, #ifdef, #ifndef) | *4* | ✔ | |
| 8 Preprocessor statements other than #if, #ifdef, and #ifndef | *4* | ✔ | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |

Figure 5-2 Definition for Logical Source Statements, Page 3

| Definition name: ***Logical Source Statements*** ***(basic definition)*** | | Includes | Excludes |
|---|---|---|---|
| **CMS-2** **Listed elements are assigned to** | | | |
| 1 Keywords like SYS-PROC and SYS-DD **statement type –>** | ***3*** | ✔ | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **COBOL** | | | |
| 1 “PROCEDURE DIVISION”, “END DECLARATIVES”, etc. | ***3*** | ✔ | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **FORTRAN** | | | |
| 1 END statements | ***1*** | ✔ | |
| 2 Format statements | ***3*** | ✔ | |
| 3 Entry statements | ***3*** | ✔ | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| **JOVIAL** | | | |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| **Pascal** | | | |
| 1 Executable statements not terminated by semicolons | ***1*** | ✔ | |
| 2 Keywords like INTERFACE and IMPLEMENTATION | ***3*** | ✔ | |
| 3 FORWARD declarations | ***3*** | ✔ | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |

Figure 5-2 Definition for Logical Source Statements, Page 4

| Definition name: ***Logical Source Statements (basic definition)*** | | Includes | Excludes |
|---|---|---|---|
| **Listed elements are assigned to statement type –>** | | | |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |

**Summary of Statement Types**

**Executable statements**

Executable statements cause runtime actions. They may be simple statements such as assignments, goto's, procedure calls, macro calls, returns, breaks, exits, stops, continues, nulls, no-ops, empty statements, and FORTRAN's END. Or they may be structured or compound statements, such as conditional statements, repetitive statements, and "with" statements. Languages like Ada, C, C++, and Pascal have block statements [begin…end and {…}] that are classified as executable when used where other executable statements would be permitted. C and C++ define expressions as executable statements when they terminate with a semicolon, and C++ has a <declaration> statement that is executable.

**Declarations**

Declarations are nonexecutable program elements that affect an assembler's or compiler's interpretation of other program elements. They are used to name, define, and initialize; to specify internal and external interfaces; to assign ranges for bounds checking; and to identify and bound modules and sections of code. Examples include declarations of names, numbers, constants, objects, types, subtypes, programs, subprograms, tasks, exceptions, packages, generics, macros, and deferred constants. Declarations also include renaming declarations, use clauses, and declarations that instantiate generics. Mandatory begin…end and {…} symbols that delimit bodies of programs and subprograms are integral parts of program and subprogram declarations. Language superstructure elements that establish boundaries for different sections of source code are also declarations. Examples include terms such as PROCEDURE DIVISION, DATA DIVISION, DECLARATIVES, END DECLARATIVES, INTERFACE, IMPLEMENTATION, SYS-PROC, and SYS-DD. Declarations, in general, are never required by language specifications to initiate runtime actions, although some languages permit compilers to implement them that way.

**Compiler Directives**

Compiler directives instruct compilers, preprocessors, or translators (but not runtime systems) to perform special actions. Some, such as Ada's pragma and COBOL's COPY, REPLACE, and USE, are integral parts of the source language. In other languages like C and C++, special symbols like # are used along with standardized keywords to direct preprocessor or compiler actions. Still other languages rely on nonstandardized methods supplied by compiler vendors. In these languages, directives are often designated by special symbols such as #, $, and {$}.

Figure 5-2 Definition for Logical Source Statements, Page 5

## 5.3. Data Specifications—Getting Data for Special Purposes

The next seven examples show how the checklist for source code size can be used to define rules for capturing different sets of data:

- Data Spec A (project tracking): an example that asks for the size data of the sort you should be using to track development status.
- Data Spec B (project analysis): an example that asks for end-of-project data you can use to improve future estimates.
- Data Spec C (reuse measurement): an example that asks for data that you can use to evaluate the extent of software reuse.
- Data Spec B+C (project analysis): an example that shows how two data specifications can sometimes be combined.
- Dead code counting: an example that shows how you can use the checklist to define the rules used to collect separate counts for dead code.
- Relaxing the rule of mutual exclusivity: an example of a case where double counting can be useful.
- Comment subtypes: an example that shows how you can use the checklist to get detailed data on subsets of attribute values

In presenting these examples, an important point deserves emphasis: Although the measurement and recording requirements for the examples differ, the definition of software size remains constant—only the supplemental data elements recorded change. Thus, the meanings of reported sizes do not change. The different sets of data simply provide additional insights into the progress or processes associated with developing and maintaining software systems.

The **Clarifications** sections on pages 3 through 5 of the checklist are not reproduced with the example data specifications in this section, since they remain the same as in the basic definitions in Figures 5-1 and 5-2. You should complete these pages and include them with your data specifications whenever the checks on these pages differ from those in the basic definition. You should also include the **Clarifications** pages whenever a checklist for the basic definition is not attached. When the **Clarifications** pages are attached, the data specification must be restricted to either physical lines or logical statements since the language-specific counting rules for the two measures will differ.

### 5.3.1. Specifying data for project tracking

The first example is our view of a useful specification for the size data we would like to have for basic project tracking. Suppose we want to track the progress of a product through each our production processes (the **How produced** attribute). To do this, we will need to make periodic measurements in which we line our size observations up against the **Development status** attribute, as illustrated in Figure 5-3.

| | coded | unit tested | integrated | test readiness review completed | CSCI tests completed | system tests completed |
|---|---|---|---|---|---|---|
| programmed | | | | | | |
| generated | | | | | | |
| converted | | | | | | |
| copied | | | | | | |
| modified | | | | | | |
| removed | | | | | | |

Figure 5-3 Example Data Array for Project Tracking

We could try to state this specification in words. Were we to do so, the results might look like this:

**Data Specification for Project Tracking (An Example)**

For each source language, measure and record these values:

- Total lines (or statements)
- A two-dimensional array showing the number of lines (statements)
    - in each development status
    - for each production class

Although this specification seems neat and logical, it is not precise enough to ensure that we will get the data we want, or that we will know what is in the data we get. For example, there is nothing in this verbal specification that says which elements of **Development status** and **How produced** are to be measured (we might want only a subset), or what rules are to be applied to the other attributes.

Figure 5-4 shows how much more explicit we can be, if we use the definition checklist to express our requirements. The two attributes to be measured are identified by checkmarks in their **Data array** boxes. The **Language** attribute also has its **Data array** box checked to show that we want separate counts for each source language. The **Definition** boxes are checked for the other attributes to show that they describe the coverage rules to be applied when counting and recording values for the **How produced** and **Development status** attributes.

## Definition Checklist for Source Statement Counts

Definition name: ***Data Spec A: Project Tracking Example (for tracking status vs. how produced)***
Date: ***8/7/92***
Originator: ***SEI***

**Measurement unit:** **Physical source lines** ☐ **Logical source statements** ☐

| **Statement type** Definition ✔ Data array ☐ | | **Includes** | **Excludes** |
|---|---|---|---|
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | |
| 1 Executable **Order of precedence ->** | 1 | ✔ | |
| 2 Nonexecutable | | | |
| 3 Declarations | 2 | ✔ | |
| 4 Compiler directives | 3 | ✔ | |
| 5 Comments | | | |
| 6 On their own lines | 4 | | ✔ |
| 7 On lines with source code | 5 | | ✔ |
| 8 Banners and nonblank spacers | 6 | | ✔ |
| 9 Blank (empty) comments | 7 | | ✔ |
| 10 Blank lines | 8 | | ✔ |
| 11 | | | |
| 12 | | | |

| **How produced** Definition ☐ Data array ✔ | **Includes** | **Excludes** |
|---|---|---|
| 1 Programmed | ✔ | |
| 2 Generated with source code generators | ✔ | |
| 3 Converted with automated translators | ✔ | |
| 4 Copied or reused without change | ✔ | |
| 5 Modified | ✔ | |
| 6 Removed | ✔ | |
| 7 | | |
| 8 | | |

| **Origin** Definition ✔ Data array ☐ | **Includes** | **Excludes** |
|---|---|---|
| 1 New work: no prior existence | ✔ | |
| 2 Prior work: taken or adapted from | | |
| 3 A previous version, build, or release | ✔ | |
| 4 Commercial, off-the-shelf software (COTS), other than libraries | ✔ | |
| 5 Government furnished software (GFS), other than reuse libraries | ✔ | |
| 6 Another product | ✔ | |
| 7 A vendor-supplied language support library (unmodified) | | ✔ |
| 8 A vendor-supplied operating system or utility (unmodified) | | ✔ |
| 9 A local or modified language support library or operating system | ✔ | |
| 10 Other commercial library | ✔ | |
| 11 A reuse library (software designed for reuse) | ✔ | |
| 12 Other software component or library | ✔ | |
| 13 | | |
| 14 | | |

| **Usage** Definition ✔ Data array ☐ | **Includes** | **Excludes** |
|---|---|---|
| 1 In or as part of the primary product | ✔ | |
| 2 External to or in support of the primary product | | ✔ |
| 3 | | |

Figure 5-4 Data Spec A (Project Tracking)

Definition name: ***Data Spec A: Project Tracking Example***
***(for tracking status vs. how produced)***

| **Delivery** — **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 Delivered | | |
| 2 Delivered as source | ✓ | |
| 3 Delivered in compiled or executable form, but not as source | ✓ | |
| 4 Not delivered | | |
| 5 Under configuration control | | ✓ |
| 6 Not under configuration control | | ✓ |
| 7 | | |

| **Functionality** — **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 Operative | ✓ | |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | |
| 3 Functional (intentional dead code, reactivated for special purposes) | ✓ | |
| 4 Nonfunctional (unintentionally present) | | ✓ |
| 5 | | |
| 6 | | |

| **Replications** — **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 Master source statements (originals) | ✓ | |
| 2 Physical replicates of master statements, stored in the master code | ✓ | |
| 3 Copies inserted, instantiated, or expanded when compiling or linking | | ✓ |
| 4 Postproduction replicates—as in distributed, redundant, | | ✓ |
| or reparameterized systems | | |
| 5 | | |

| **Development status** — **Definition** [ ] **Data array** [✓] | **Includes** | **Excludes** |
|---|---|---|
| *Each statement has one and only one status, usually that of its parent unit.* | | |
| 1 Estimated or planned | | ✓ |
| 2 Designed | | ✓ |
| 3 Coded | ✓ | |
| 4 Unit tests completed | ✓ | |
| 5 Integrated into components | ✓ | |
| 6 Test readiness review completed | ✓ | |
| 7 Software (CSCI) tests completed | ✓ | |
| 8 System tests completed | ✓ | |
| 9 | | |
| 10 | | |
| 11 | | |

| **Language** — **Definition** [ ] **Data array** [✓] | | **Includes** | **Excludes** |
|---|---|---|---|
| *List each source language on a separate line.* | | | |
| 1 | ***Separate totals for each language*** | ✓ | |
| 2 Job control languages | | | |
| 3 | | | |
| 4 Assembly languages | | | |
| 5 | | | |
| 6 Third generation languages | | | |
| 7 | | | |
| 8 Fourth generation languages | | | |
| 9 | | | |
| 10 Microcode | | | |
| 11 | | | |

Figure 5-4 Data Spec A (Project Tracking, Page 2)

Figure 5-3 showed why we use the term **Data array** when defining the elements we want measured. In Data Spec A, we want to track the progress of each production process. This means that we must get separate counts for each intersection of the **How produced** and **Development status** attributes. Without array structures for recording this data, we would be unable to distinguish progress in copying or modifying code from progress in developing new statements. As a consequence, we could be presented with overly optimistic pictures of development progress.

We can easily show that total size alone is inadequate as a progress measure—all we have to do is picture a case where substantial portions of the code are expected to be copied from pre-existing components. Figure 5-5 shows such an example. If we know only that 40%, say, of our total statements are coded, we have no way of knowing what proportion of the programming work we still have ahead of us. The inadequacy of total counts as management metrics is compounded further if, as in Figure 5-5, substantial portions of copied code turn out to be unsatisfactory and have to be replaced.

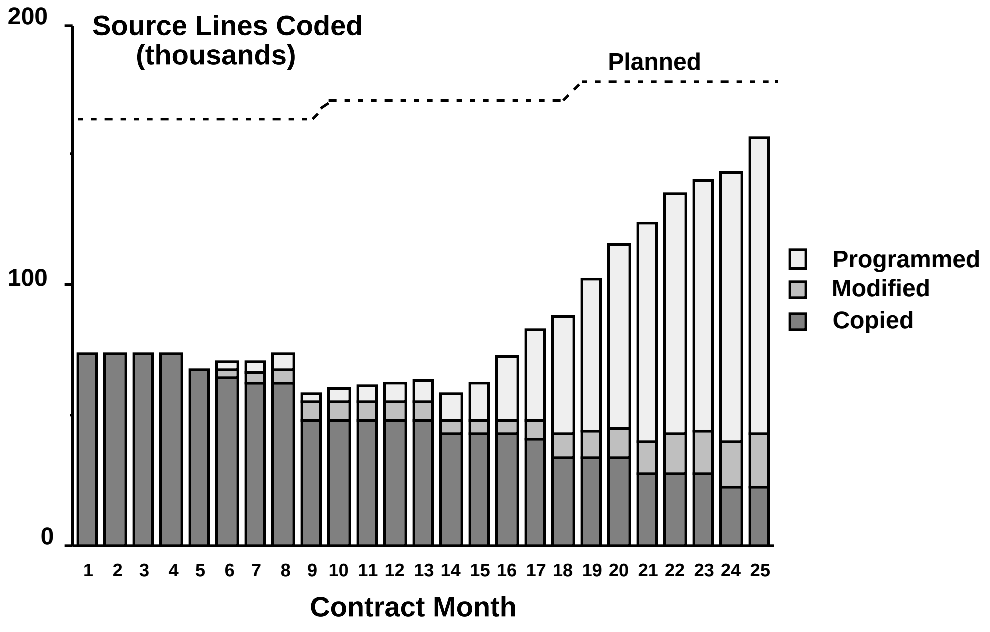


Figure 5-5 The Case of Disappearing Reuse

There is no requirement that the coverage rules for any of the attributes in a data specification be the same as those used in our basic definition for size. For example, if we want to track progress in developing software used external to or in support of our primary product, we can simply reverse the checkmarks under the **Usage** attribute. Alternatively, we can check the **Data array** box for the **Usage** attribute and then, by checking both elements under that attribute, show that we want counts for both elements included in the recorded data.

Similarly, the values included in a **Data array** need not match those included in a basic size definition. In Data Spec A, for example, several **Development status** values for which counts are requested are not included in the basic definition. Counts for removed statements were also requested, even though they are not in the basic definition for size. In the example for Data Spec B in the next subsection, counts will be requested for two different types of comments—neither of which is included in the basic definition for size.

Counts for data elements defined by data specifications are always made in addition to the counts made for a basic definition for size. The processes of counting, recording, and reporting these supplemental elements never change the basic definition.

When tracking progress, you may, if you wish, collect only subsets of Data Spec A. For example, if code removal is not labor-intensive for your project or if there is no generated or converted code, these elements (the rows in Figure 5-3) can be deleted from Data Spec A. Similarly, if a software development process has no formal test readiness review or if final software tests are system tests, the columns in Figure 5-3 for the respective **Development status** elements can also be omitted. The checklist for Data Spec A should then be modified as illustrated in Figure 5-6 to formalize these specifications and relate them to the rules that apply to other checklist attributes. The simplified data array that results would then look like Figure 5-7. Here only 12 data elements need to be counted rather than the 36 in Data Spec A. This reduced array is sufficient for gathering data for relatively simple plots like the one in Figure 5-5, where we illustrated The Case of Disappearing Reuse.

| **How produced** | **Definition** ☐ | **Data array** ✓ | **Includes** | **Excludes** |
|---|---|---|---|---|
| 1 Programmed | | | ✓ | |
| 2 Generated with source code generators | | | | ✓ |
| 3 Converted with automated translators | | | | ✓ |
| 4 Copied or reused without change | | | ✓ | |
| 5 Modified | | | ✓ | |
| 6 Removed | | | | ✓ |
| 7 | | | | |
| 8 | | | | |

| **Development status** | **Definition** ☐ | **Data array** ✓ | **Includes** | **Excludes** |
|---|---|---|---|---|
| *Each statement has one and only one status, usually that of its parent unit.* | | | ▨ | ▨ |
| 1 Estimated or planned | | | | ✓ |
| 2 Designed | | | | ✓ |
| 3 Coded | | | ✓ | |
| 4 Unit tests completed | | | ✓ | |
| 5 Integrated into components | | | ✓ | |
| 6 Test readiness review completed | | | | ✓ |
| 7 Software (CSCI) tests completed | | | | ✓ |
| 8 System tests completed | | | ✓ | |
| 9 | | | | |
| 10 | | | | |
| 11 | | | | |

Figure 5-6 Specification for a Simplified Data Array

|  | coded | unit tested | integrated | system tests completed |
|---|---|---|---|---|
| programmed |  |  |  |  |
| copied |  |  |  |  |
| modified |  |  |  |  |

Figure 5-7 Simplified Data Array for Project Tracking

The checklist that we completed for Data Spec A and the examples that follow apply equally to physical source lines and logical source statements. We have left the **Measurement unit** designations at the top of the forms blank so that you can state your own preferences. When we have automated counters that can identify both physical lines and logical statements, we will probably elect to use both measures—the additional costs will not be high.

### 5.3.2. Specifying data for project analysis

Our second example addresses project analysis. It asks for data that we can use to improve our estimation and planning for future products. We would not normally collect this data while development is in progress. Instead, we would use this specification at the end of the project to ensure that we get the data that we need for calibrating cost models and for sizing future projects. An outline of this data specification is shown verbally below and schematically in Figure 5-8. Figure 5-9 presents a checklist that makes the measurement requirements specific and relates them to the rules that apply to other attributes.

**Data Specification for Project Analysis (An Example)**

For each source language, measure and record these values:

- Total lines (or statements)
- A two-dimensional array showing the number of lines (statements)
    - in each statement type
    - for each production class

| | executable statements | declarations | compiler directives | comments on their own lines | comments on lines with source lines |
|---|---|---|---|---|---|
| programmed | | | | | |
| generated | | | | | |
| converted | | | | | |
| copied | | | | | |
| modified | | | | | |
| removed | | | | | |

Figure 5-8 Example Data Array for Project Analysis

Organizations just starting to use formal definitions to capture this kind of information may want to begin with a somewhat simpler array. For example, if your tools do not yet support capturing this information automatically, you may want to consider deleting the request for counts of **removed** statements under the **How produced** attribute. You may also want to omit the columns for classifying and counting comments.

If your counting tools are automated and if they are capable of distinguishing among different statement types, you may want to consider adding development status to these data requirements and collecting the data periodically during development. The additional costs for periodic measurement (compared to Data Spec A) will not be great, although database storage requirements will increase. Your real difficulties will come in summarizing and presenting the data in ways that people can read. But at least you will have the data should you want it for addressing special needs. For example, in Ada developments, significant amounts of nonexecutable code can be generated during design via package specifications and compilable design languages. Periodic reporting of **Statement type** versus **How produced** would give insight into the transition of these efforts from design into the implementation of executable code.

## Definition Checklist for Source Statement Counts

Definition name: ***Data Spec B: Project Analysis Example*** Date: ***8/7/92***
***(end-of-project data used to improve future estimates)*** Originator: ***SEI***

**Measurement unit:** **Physical source lines** ☐ **Logical source statements** ☐

| **Statement type** Definition ☐ Data array ✓ | Order | **Includes** | **Excludes** |
|---|---|---|---|
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | |
| 1 Executable **Order of precedence ->** | 1 | ✓ | |
| 2 Nonexecutable | | | |
| 3 Declarations | 2 | ✓ | |
| 4 Compiler directives | 3 | ✓ | |
| 5 Comments | | | |
| 6 On their own lines | 4 | ✓ | |
| 7 On lines with source code | 5 | ✓ | |
| 8 Banners and nonblank spacers | 6 | | ✓ |
| 9 Blank (empty) comments | 7 | | ✓ |
| 10 Blank lines | 8 | | ✓ |
| 11 | | | |
| 12 | | | |

| **How produced** Definition ☐ Data array ✓ | **Includes** | **Excludes** |
|---|---|---|
| 1 Programmed | ✓ | |
| 2 Generated with source code generators | ✓ | |
| 3 Converted with automated translators | ✓ | |
| 4 Copied or reused without change | ✓ | |
| 5 Modified | ✓ | |
| 6 Removed | ✓ | |
| 7 | | |
| 8 | | |

| **Origin** Definition ✓ Data array ☐ | **Includes** | **Excludes** |
|---|---|---|
| 1 New work: no prior existence | ✓ | |
| 2 Prior work: taken or adapted from | | |
| 3 A previous version, build, or release | ✓ | |
| 4 Commercial, off-the-shelf software (COTS), other than libraries | ✓ | |
| 5 Government furnished software (GFS), other than reuse libraries | ✓ | |
| 6 Another product | ✓ | |
| 7 A vendor-supplied language support library (unmodified) | | ✓ |
| 8 A vendor-supplied operating system or utility (unmodified) | | ✓ |
| 9 A local or modified language support library or operating system | ✓ | |
| 10 Other commercial library | ✓ | |
| 11 A reuse library (software designed for reuse) | ✓ | |
| 12 Other software component or library | ✓ | |
| 13 | | |
| 14 | | |

| **Usage** Definition ✓ Data array ☐ | **Includes** | **Excludes** |
|---|---|---|
| 1 In or as part of the primary product | ✓ | |
| 2 External to or in support of the primary product | | ✓ |
| 3 | | |

Figure 5-9 Data Spec B (Project Analysis)

Definition name: ***Data Spec B: Project Analysis Example (end-of-project data used to improve future estimates)***

| **Delivery**   **Definition** ✓   **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|
| 1 Delivered | | |
| 2 Delivered as source | ✓ | |
| 3 Delivered in compiled or executable form, but not as source | ✓ | |
| 4 Not delivered | | |
| 5 Under configuration control | | ✓ |
| 6 Not under configuration control | | ✓ |
| 7 | | |

| **Functionality**   **Definition** ✓   **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|
| 1 Operative | ✓ | |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | |
| 3 Functional (intentional dead code, reactivated for special purposes) | ✓ | |
| 4 Nonfunctional (unintentionally present) | | ✓ |
| 5 | | |
| 6 | | |

| **Replications**   **Definition** ✓   **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|
| 1 Master source statements (originals) | ✓ | |
| 2 Physical replicates of master statements, stored in the master code | ✓ | |
| 3 Copies inserted, instantiated, or expanded when compiling or linking | | ✓ |
| 4 Postproduction replicates—as in distributed, redundant, or reparameterized systems | | ✓ |
| 5 | | |

| **Development status**   **Definition** ✓   **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|
| *Each statement has one and only one status, usually that of its parent unit.* | | |
| 1 Estimated or planned | | ✓ |
| 2 Designed | | ✓ |
| 3 Coded | | ✓ |
| 4 Unit tests completed | | ✓ |
| 5 Integrated into components | | ✓ |
| 6 Test readiness review completed | | ✓ |
| 7 Software (CSCI) tests completed | | ✓ |
| 8 System tests completed | ✓ | |
| 9 | | |
| 10 | | |
| 11 | | |

| **Language**   **Definition** ☐   **Data array** ✓ | | **Includes** | **Excludes** |
|---|---|---|---|
| *List each source language on a separate line.* | | | |
| 1 | ***Separate totals for each language*** | ✓ | |
| 2 Job control languages | | | |
| 3 | | | |
| 4 Assembly languages | | | |
| 5 | | | |
| 6 Third generation languages | | | |
| 7 | | | |
| 8 Fourth generation languages | | | |
| 9 | | | |
| 10 Microcode | | | |
| 11 | | | |

Figure 5-9 Data Spec B (Project Analysis, Page 2)

### 5.3.3. Specifying data for reuse measurement

The third example asks for data elements that will help us quantify and interpret software reuse. These same data elements can also be used to help us compute and interpret derived measures of productivity and quality. The measurement requirements for the example are outlined below. Figure 5-10 shows a checklist that makes these requirements specific and relates them to the rules that apply to other attributes.

**Data Specification for Reuse Measurement (An Example)**

For each source language, measure and record these values:

- Total lines (or statements)
- A two-dimensional array showing the number of lines (statements)
  - in each production class
  - for each origin

Collecting data for Data Spec C is likely to require more effort and more configuration management support than in the previous examples. Tracing origins of completed statements is inherently difficult. It almost always requires tagging each physical line to indicate its origin or separately passing code from each origin through a statement counter. Except in fairly simple cases, you may want to defer tracing origins of statements until you have tools and automated practices in place to support collecting this data.

# Definition Checklist for Source Statement Counts

Definition name: ***Data Spec C: Reuse Measurement Example*** Date: ***8/7/92***

Originator: ***SEI***

| | | |
|---|---|---|
| **Measurement unit:** | **Physical source lines** | ☐ |
| | **Logical source statements** | ☐ |

| **Statement type** | **Definition** ✓ **Data array** ☐ | | **Includes** | **Excludes** |
|---|---|---|---|---|
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | | |
| 1 Executable | **Order of precedence ->** | 1 | ✓ | |
| 2 Nonexecutable | | | | |
| 3 Declarations | | 2 | ✓ | |
| 4 Compiler directives | | 3 | ✓ | |
| 5 Comments | | | | |
| 6 On their own lines | | 4 | | ✓ |
| 7 On lines with source code | | 5 | | ✓ |
| 8 Banners and nonblank spacers | | 6 | | ✓ |
| 9 Blank (empty) comments | | 7 | | ✓ |
| 10 Blank lines | | 8 | | ✓ |
| 11 | | | | |
| 12 | | | | |

| **How produced** | **Definition** ☐ **Data array** ✓ | **Includes** | **Excludes** |
|---|---|---|---|
| 1 Programmed | | ✓ | |
| 2 Generated with source code generators | | ✓ | |
| 3 Converted with automated translators | | ✓ | |
| 4 Copied or reused without change | | ✓ | |
| 5 Modified | | ✓ | |
| 6 Removed | | ✓ | |
| 7 | | | |
| 8 | | | |

| **Origin** | **Definition** ☐ **Data array** ✓ | **Includes** | **Excludes** |
|---|---|---|---|
| 1 New work: no prior existence | | ✓ | |
| 2 Prior work: taken or adapted from | | | |
| 3 A previous version, build, or release | | ✓ | |
| 4 Commercial, off-the-shelf software (COTS), other than libraries | | ✓ | |
| 5 Government furnished software (GFS), other than reuse libraries | | ✓ | |
| 6 Another product | | ✓ | |
| 7 A vendor-supplied language support library (unmodified) | | | ✓ |
| 8 A vendor-supplied operating system or utility (unmodified) | | | ✓ |
| 9 A local or modified language support library or operating system | | ✓ | |
| 10 Other commercial library | | ✓ | |
| 11 A reuse library (software designed for reuse) | | ✓ | |
| 12 Other software component or library | | ✓ | |
| 13 | | | |
| 14 | | | |

| **Usage** | **Definition** ✓ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|
| 1 In or as part of the primary product | | ✓ | |
| 2 External to or in support of the primary product | | | ✓ |
| 3 | | | |

Figure 5-10 Data Spec C (Reuse Measurement)

Definition name: ***Data Spec C: Reuse Measurement Example***

| **Delivery** | **Definition** ✓ | **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|---|
| 1 Delivered | | | | |
| 2 Delivered as source | | | ✓ | |
| 3 Delivered in compiled or executable form, but not as source | | | ✓ | |
| 4 Not delivered | | | | |
| 5 Under configuration control | | | | ✓ |
| 6 Not under configuration control | | | | ✓ |
| 7 | | | | |

| **Functionality** | **Definition** ✓ | **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|---|
| 1 Operative | | | ✓ | |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | | | |
| 3 Functional (intentional dead code, reactivated for special purposes) | | | ✓ | |
| 4 Nonfunctional (unintentionally present) | | | | ✓ |
| 5 | | | | |
| 6 | | | | |

| **Replications** | **Definition** ✓ | **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|---|
| 1 Master source statements (originals) | | | ✓ | |
| 2 Physical replicates of master statements, stored in the master code | | | ✓ | |
| 3 Copies inserted, instantiated, or expanded when compiling or linking | | | | ✓ |
| 4 Postproduction replicates—as in distributed, redundant, or reparameterized systems | | | | ✓ |
| 5 | | | | |

| **Development status** | **Definition** ✓ | **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|---|
| *Each statement has one and only one status, usually that of its parent unit.* | | | | |
| 1 Estimated or planned | | | | ✓ |
| 2 Designed | | | | ✓ |
| 3 Coded | | | | ✓ |
| 4 Unit tests completed | | | | ✓ |
| 5 Integrated into components | | | | ✓ |
| 6 Test readiness review completed | | | | ✓ |
| 7 Software (CSCI) tests completed | | | | ✓ |
| 8 System tests completed | | | ✓ | |
| 9 | | | | |
| 10 | | | | |
| 11 | | | | |

| **Language** | **Definition** ☐ | **Data array** ✓ | **Includes** | **Excludes** |
|---|---|---|---|---|
| *List each source language on a separate line.* | | | | |
| 1 | ***Separate totals for each language*** | | ✓ | |
| 2 Job control languages | | | | |
| 3 | | | | |
| 4 Assembly languages | | | | |
| 5 | | | | |
| 6 Third generation languages | | | | |
| 7 | | | | |
| 8 Fourth generation languages | | | | |
| 9 | | | | |
| 10 Microcode | | | | |
| 11 | | | | |

Figure 5-10 Data Spec C (Reuse Measurement, Page 2)

### 5.3.4. Combining data specifications

Different data specifications are not necessarily mutually exclusive. In many instances, they can be combined. Figure 5-11 is an example, showing the combination of Data Specs B and C. Combinations like these have the benefit of reducing the number of sheets of paper that data collectors must work with.

There are costs, however, as well as operational hurdles that must be overcome when combining data specifications. One cost is that database requirements are increased (in the B+C case) from the maintenance of two arrays, one 5x9 and the other 5x5, to the maintenance of a three-dimensional array that has 5x5x9 elements. This is a factor of three increase in data storage. Another cost is that separate passes through the counter may have to be made to get counts for attribute classes such as **Origins** that can be separated only by manual means.

The operational hurdle is that organizations may experience difficulty transmitting highly multidimensional measurement results from the point of collection to the point of database entry. Until you have automated tools to aid in this process, you will probably be wise to limit data recording to one or two dimensions (attributes) at a time for each source language. Note, however, that there is no limit to the number of one- and two-dimensional data specifications you can use.

Figure 5-11 shows how data specifications can be combined. Combining data specifications may be useful when organizations have automated tools to support measurement, recording, and database entry.

## Definition Checklist for Source Statement Counts

Definition name: ***Data Spec B+C: Project Analysis (combined specifications)***   Date: ***8/7/92***

Originator: ***SEI***

| | | | |
|---|---|---|---|
| **Measurement unit:** | **Physical source lines** ☐ | | |
| | **Logical source statements** ☐ | | |
| **Statement type** — **Definition** ☐ **Data array** ✔ | | **Includes** | **Excludes** |
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | |
| 1 Executable | **Order of precedence ->** 1 | ✔ | |
| 2 Nonexecutable | | | |
| 3 Declarations | 2 | ✔ | |
| 4 Compiler directives | 3 | ✔ | |
| 5 Comments | | | |
| 6 On their own lines | 4 | ✔ | |
| 7 On lines with source code | 5 | ✔ | |
| 8 Banners and nonblank spacers | 6 | | ✔ |
| 9 Blank (empty) comments | 7 | | ✔ |
| 10 Blank lines | 8 | | ✔ |
| 11 | | | |
| 12 | | | |
| **How produced** — **Definition** ☐ **Data array** ✔ | | **Includes** | **Excludes** |
| 1 Programmed | | ✔ | |
| 2 Generated with source code generators | | ✔ | |
| 3 Converted with automated translators | | ✔ | |
| 4 Copied or reused without change | | ✔ | |
| 5 Modified | | ✔ | |
| 6 Removed | | ✔ | |
| 7 | | | |
| 8 | | | |
| **Origin** — **Definition** ☐ **Data array** ✔ | | **Includes** | **Excludes** |
| 1 New work: no prior existence | | ✔ | |
| 2 Prior work: taken or adapted from | | | |
| 3 A previous version, build, or release | | ✔ | |
| 4 Commercial, off-the-shelf software (COTS), other than libraries | | ✔ | |
| 5 Government furnished software (GFS), other than reuse libraries | | ✔ | |
| 6 Another product | | ✔ | |
| 7 A vendor-supplied language support library (unmodified) | | | ✔ |
| 8 A vendor-supplied operating system or utility (unmodified) | | | ✔ |
| 9 A local or modified language support library or operating system | | ✔ | |
| 10 Other commercial library | | ✔ | |
| 11 A reuse library (software designed for reuse) | | ✔ | |
| 12 Other software component or library | | ✔ | |
| 13 | | | |
| 14 | | | |
| **Usage** — **Definition** ✔ **Data array** ☐ | | **Includes** | **Excludes** |
| 1 In or as part of the primary product | | ✔ | |
| 2 External to or in support of the primary product | | | ✔ |
| 3 | | | |

Figure 5-11 Data Spec B+C (Combined Specifications)

Definition name: ***Data Spec B+C: Project Analysis (combined specifications)***

| | | | | | |
|---|---|---|---|---|---|
| **Delivery** | **Definition** | ✓ | **Data array** | ☐ | **Includes** | **Excludes** |
| 1 Delivered | | | | | ▨ | ▨ |
| 2 Delivered as source | | | | | ✓ | |
| 3 Delivered in compiled or executable form, but not as source | | | | | ✓ | |
| 4 Not delivered | | | | | ▨ | ▨ |
| 5 Under configuration control | | | | | | ✓ |
| 6 Not under configuration control | | | | | | ✓ |
| 7 | | | | | | |
| **Functionality** | **Definition** | ✓ | **Data array** | ☐ | **Includes** | **Excludes** |
| 1 Operative | | | | | ✓ | |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | | | | ▨ | ▨ |
| 3 Functional (intentional dead code, reactivated for special purposes) | | | | | ✓ | |
| 4 Nonfunctional (unintentionally present) | | | | | | ✓ |
| 5 | | | | | | |
| 6 | | | | | | |
| **Replications** | **Definition** | ✓ | **Data array** | ☐ | **Includes** | **Excludes** |
| 1 Master source statements (originals) | | | | | ✓ | |
| 2 Physical replicates of master statements, stored in the master code | | | | | ✓ | |
| 3 Copies inserted, instantiated, or expanded when compiling or linking | | | | | | ✓ |
| 4 Postproduction replicates—as in distributed, redundant, | | | | | | ✓ |
| or reparameterized systems | | | | | | |
| 5 | | | | | | |
| **Development status** | **Definition** | ✓ | **Data array** | ☐ | **Includes** | **Excludes** |
| *Each statement has one and only one status, usually that of its parent unit.* | | | | | ▨ | ▨ |
| 1 Estimated or planned | | | | | | ✓ |
| 2 Designed | | | | | | ✓ |
| 3 Coded | | | | | | ✓ |
| 4 Unit tests completed | | | | | | ✓ |
| 5 Integrated into components | | | | | | ✓ |
| 6 Test readiness review completed | | | | | | ✓ |
| 7 Software (CSCI) tests completed | | | | | | ✓ |
| 8 System tests completed | | | | | ✓ | |
| 9 | | | | | | |
| 10 | | | | | | |
| 11 | | | | | | |
| **Language** | **Definition** | ☐ | **Data array** | ✓ | **Includes** | **Excludes** |
| *List each source language on a separate line.* | | | | | ▨ | ▨ |
| 1 | ***Separate totals for each language*** | | | | ✓ | |
| 2 Job control languages | | | | | | |
| 3 | | | | | | |
| 4 Assembly languages | | | | | | |
| 5 | | | | | | |
| 6 Third generation languages | | | | | | |
| 7 | | | | | | |
| 8 Fourth generation languages | | | | | | |
| 9 | | | | | | |
| 10 Microcode | | | | | | |
| 11 | | | | | | |

Figure 5-11 Data Spec B+C (Combined Specifications, Page 2)

## 5.3.5. Counting dead code

In practice, we would augment our definitions with a specification for one additional data element—a request to measure and report the total amount of dead code included in the overall measure for size. This would help us assess the reliability of the size counts we receive, and it would help us evaluate the effectiveness and progress of our dead code removal practices.

We believe that the best way to focus attention on actions needed to eliminate potentially dangerous contaminations in delivered products is to attempt to measure the contaminating elements. Since dead code is an excluded element in our definition for total size and since we see little benefit in mapping its characteristics in detailed arrays against other attributes, we would make this request as a supplement to our primary size definition. We would convey this request by means of a checklist that is identical to our definition checklist for total size, except that the rules for the **Functionality** attribute would be as shown in Figure 5-12. We would also ask that the steps taken to identify and exclude inoperative code be reported in a supplemental rules form. We will illustrate such a form in Figure 7-3 of Chapter 7.

| **Functionality** **Definition** ☐ **Data array** ✓ | **Includes** | **Excludes** |
|---|---|---|
| 1 Operative | ✓ | |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | ▨ | ▨ |
| 3 Functional (intentional dead code, reactivated for special purposes) | ✓ | |
| 4 Nonfunctional (unintentionally present) | ✓ | |
| 5 | | |
| 6 | | |

Figure 5-12 Dead Code Data Spec

## 5.3.6. Relaxing the rule of mutual exclusivity

In Section 3-2, we discussed the reasons for keeping the values within attributes mutually exclusive. There is one useful exception where the guidelines for mutual exclusivity can be safely ignored and values can be permitted to overlap without danger of miscounting. This occurs when the values are used not to define elements included in or excluded from a measure, but are used only as part of a data specification that identifies individual data elements to be collected and reported separately.

To give just one example, suppose you want to obtain counts for textual comments, irrespective of whether you include comments in your overall measure for source code size. One easy way to specify this is to insert the phrase "Textual comments: #6 + #7, even if on lines with other statements" on one of the blank lines under the **Statement type** attribute. Then, by checking the **Data array** box, you can indicate that you want counts for these comments to be collected along with your other counts. If only simple counts (counts not arrayed with other data) are needed, the best way to get them is to use a fresh checklist to create a separate data specification.

Figure 5-13 is an example of how you can modify the checklist to get the counts you want. Here counts for textual comments are requested in addition to counts for executable statements, declarations, and compiler directives. If the counts for the other statement types are not needed, you can move their checkmarks to the **Excludes** column.

| | **Statement type** | **Definition** ☐ | **Data array** ✔ | | **Includes** | **Excludes** |
|---|---|---|---|---|---|---|
| | *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | | ▨ | ▨ |
| 1 | Executable | | **Order of precedence ->** | 1 | ✔ | |
| 2 | Nonexecutable | | | ▨ | ▨ | ▨ |
| 3 | Declarations | | | 2 | ✔ | |
| 4 | Compiler directives | | | 3 | ✔ | |
| 5 | Comments | | | ▨ | ▨ | ▨ |
| 6 | On their own lines | | | 4 | | ✔ |
| 7 | On lines with source code | | | 5 | | ✔ |
| 8 | Banners and nonblank spacers | | | 6 | | ✔ |
| 9 | Blank (empty) comments | | | 7 | | ✔ |
| 10 | Blank lines | | | 8 | | ✔ |
| 11 | ***Textual comments: #6 + #7, even if on lines with other*** | | | — | ✔ | |
| 12 | ***statements*** | | | | | |

Figure 5-13 Specification for Counts of Textual Comments

### 5.3.7. Asking for lower level details—subsets of attribute values

Some users of measurement results may want reports for attribute values that are subsets of those in the original checklist. For example, one reviewer has observed that if XYZ is an attribute value that is already in the checklist, he might like to get data on a subset of this group—say XYZ.abc—and that it should not cause great problems if he were to add such an element to the checklist. We agree, but suggest that one simple step can avoid possible confusion. Merely replace the XYZ element with two elements—XYZ.abc and XYZ.other—so that all outcomes are covered and the list remains mutually exclusive. This subsetting tactic is, in effect, one we have used already when listing values for comment types and prior work origins in the definition checklist

Figure 5-14 is an example of how we might use the definition checklist to get counts for subsets of a data element. This example splits the statement type *Comments on their own lines* into two subtypes: *Comments on their own lines (revision histories)* and *Comments on their own lines (other).* We would use a specification like this to get separate counts for the number of source lines used to record revision histories.

**Definition Checklist for Source Statement Counts**

Definition name: ***Data Spec for Comment Subtypes*** Date: ***9/1/92***

Originator: ***J. Smith***

| | | | | |
|---|---|---|---|---|
| **Measurement unit:** | **Physical source lines** | ✓ | | |
| | **Logical source statements** | | | |

| **Statement type** | **Definition** ☐ **Data array** ✓ | | **Includes** | **Excludes** |
|---|---|---|---|---|
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | | |
| 1 Executable | **Order of precedence ->** | 1 | ✓ | |
| 2 Nonexecutable | | | | |
| 3 Declarations | | 2 | ✓ | |
| 4 Compiler directives | | 3 | ✓ | |
| 5 Comments | | | | |
| 6 ~~On their own lines~~ | | 4 | — | — |
| 7 On lines with source code | ***(count these separately)*** | — | ✓ | |
| 8 Banners and nonblank spacers | | 6 | | ✓ |
| 9 Blank (empty) comments | | 7 | | ✓ |
| 10 Blank lines | | 8 | | ✓ |
| 11 ***Comments on their own lines (revision histories)*** | | ***4*** | ✓ | |
| 12 ***Comments on their own lines (othe*** | | ***5*** | ✓ | |

Figure 5-14 Getting Data on Subsets of Attribute Values

## 5.4. Other Definitions and Data Specifications

The examples in this chapter have been predominantly ones we would use for cost estimating and project tracking. When you have other reasons for measuring size, you may want alternative definitions or data specifications. For example, if you want to measure statement reuse so that you can evaluate trends in using generics, macros, or include files, you may want to include copies of inserted, instantiated, and expanded code in your statement counts. Similarly, if you want to examine processes that develop nondelivered code, you may want to count nondelivered statements. All these situations can be described with the definition checklist.

Our advice in situations where alternative definitions seem to be needed is to determine first whether you can get the information you want by using a data specification rather than by constructing a new definition. In most cases, you can. The advantage of data specifications is that they keep definitions of size from proliferating. If you can get what you need with a data specification, you can maintain a common definition of size that can be used across many projects. Then trends can be examined, and lessons learned in one place can be applied in others.

## 5.5. Other Measures

The examples in this chapter have been exclusively source statement measures. Appendix C discusses our reasons for narrowing our scope to these measures. The fact that we have selected two particular measures with which to illustrate our framework for defining software size does not imply that other measures may not be equally good or even better. Given time, we would like to apply our checklist-based methods to measures like source units (CSUs), requirements, pages of design specification, noncommentary source words, and possibly even function points or feature points. These measures and many others are mentioned in Appendix B, where we list well over 100 possible measures that could be used to describe the size of software products and activities.

# 6. Defining the Counting Unit

We now turn our attention to the question of how to recognize a source statement when we see one. Whenever either logical source statements or physical source lines are counted, rules must be established for determining exactly when statements begin and end. These rules must be complete and noncontradictory, and they must be clearly and explicitly stated. In particular, they must be precise to the point that they can be programmed into automated line and statement counters without further interpretation.

## 6.1. Statement Delimiters

Statement delimiters are the symbols and rules that identify when source statements begin and end. Explicit and unambiguous specification of all statement delimiters and delimiting rules is required if meaningful size measurements are to be obtained. Specifications for delimiters and delimiting rules are not complete until all special cases and exceptions are accounted for.

## 6.2. Logical Source Statements

Counts of logical source statements (or instructions) are attempts to measure the size of software in ways that are independent of the physical form in which the instructions appear. The thesis behind counting logical source statements is that the number of logical instructions is in some sense proportional to the amount of logic in the code. A second thesis (or hypothesis) is that effort and, hence, cost are proportional to the total logic that must be specified, designed, coded, and tested.

One *advantage* of counts of logical source statements is that they are generally less dependent on programming style than counts of physical source statements. In particular, logical counts are usually unaffected by passing source code through a source code formatter or pretty printer. Another advantage is that logical source statements are independent of naming practices—lengthy (some call them descriptive) names do not inflate statement counts.

One *disadvantage* of counts of logical statements is that interpretations are strongly dependent on the rules used for deciding what constitutes a statement and for identifying statement delimiters. This has made it difficult to ensure that consistent rules are applied across organizations. Another disadvantage is that forms and classes of statements and delimiters vary from language to language. This means that even with consistent definitions for inclusion and exclusion, counts of logical source statements may not be comparable across different source languages.

When delimiters are not specified explicitly, those who count statements can make up their own rules, and users of the results never know for sure what the measurements mean. Specifying delimiters for logical source statements requires painstaking and sometimes arduous work. For example:

- Rules for recognizing statement delimiters must address all special cases. Examples where rules must be explicit include identifying embedded statements, recognizing and evaluating continuation statements, identifying embedded literals and embedded comments, and responding to situations where more than one type of delimiter can be used.
- Many languages have more than one type of delimiter. For instance, Ada, C, C++, Pascal, and Modula-2 use different delimiters for comments than they do for declarations or executable statements.
- Not all delimiters are statement delimiters. The Ada language reference manual, for example, lists 26 symbols as delimiters [Ada 83]. Ada also uses other symbols not on the list (double dash and end-of-line) to delimit its comments.
- Statement delimiters are difficult to specify unambiguously. For example, "count terminal semicolons" expresses a concept but leaves numerous details open for interpretation. Rules like this are unsatisfactory as counting specifications. Since all semicolons look alike, responsibility is effectively dodged by the rule makers, leaving discretion to local organizations to choose among alternatives for interpreting and implementing the meaning of "terminal." This leads to the collection of statement counts of uncertain content and to untrustworthy comparability among different organizations who report size measures.
- Some languages draw little distinction between expressions and statements. C and C++ are two examples. These languages present particularly thorny cases when defining delimiters for counting logical source statements. Where other languages permit only expressions, C and C++ programmers may, if they wish, use either statements or expressions, often to nearly the same effect. You must give careful thought to the rules for identifying delimiters and statements when your objective is to make logical counts for C and C++ programs comparable to logical counts for other languages.
- Similarities between expressions and logical statements raise other questions. For example, should expressions such as those in calling arguments or logical tests be treated as logical statements? Languages such as Pascal, C, and C++ permit extensive formatting and computational instructions to be buried as expressions in *write* or *printf* statements. Should these instructions be counted as statements? If so, what are their delimiters? FORTRAN's *WRITE* statements can also contain expressions (instructions?) that produce significant amounts of computation at run time. These are all issues that have to be settled before definitions and specifications for logical source statement delimiters can be considered complete.
- Runtime debuggers often identify logical expressions used in conditional tests as executable statements and include them in their sequential statement count. This helps when single stepping is used to locate programming errors. If organizations

want to follow similar rules when creating a definition, they must state their intentions and specify the delimiters (or other rules) that they will use to identify and count these expressions.

- Those who count logical source statements by means of separators such as semicolons must specify the rules for addressing the special cases of the first and last statements in bodies and blocks of code. For example, the final statement within a Pascal block requires no punctuation, and the final statement in a program or unit ends with a period, not a semicolon. Also, in versions of Pascal that have separately compilable units, executable statements do not occur until an *implementation* statement is encountered. Other languages can have similar characteristics.
- In some languages that use statement separators rather than statement terminators (e.g., Pascal), there are many instances in which the separator is optional. Here simplistic rules such as "count semicolons" would leave the definition of *logical statement* to the whims of individual programmers. This, in the absence of strict coding standards, could destroy all hopes of comparability, even across modules within a common project.
- Specifications for logical statement delimiters must deal with all cases in which symbols used as delimiters can be used for other purposes as well. For example, semicolons in some languages can be used not only to terminate or separate statements, but also to terminate or separate expressions, function arguments, and data list elements. In addition, delimiting symbols can appear as literals in text strings and comments without performing delimiting roles.
- Counting philosophies can be influenced and complicated by the fact that many languages reserve the word *statement* to describe just executable statements. Ada, Pascal, C, C++, and Modula-2 are examples. In these languages, declarations, compiler directives, and comments are not classified as statements, they are merely declarations, compiler directives, and comments.
- Some languages (Pascal, C, C++, and Modula-2, for example) have special symbols for the beginnings and endings of comments that permit comments to flow over several physical lines. In these languages, a single pair of symbols may bracket several sentences that people may perceive as statements. Other languages (Ada, FORTRAN, and assembly, for example) terminate comments at the first end-of-line encountered. In these cases, each comment may capture but a fraction of the textual statements that programmers make. It is not at all clear that a rational system exists to resolve these differences, other than resorting to counts of physical lines when the volume of comments is of interest.

People can easily disagree on even simple examples of logical source counts. For instance, how many logical statements do you think should be counted in this code fragment?

**if** A **then** B **else** C **endif;**

We have posed this question to several audiences and have received a variety of responses. Figure 6-1 shows a summary from two audiences we polled. Clearly, people have differing views as to what they perceive logical source statements to be.

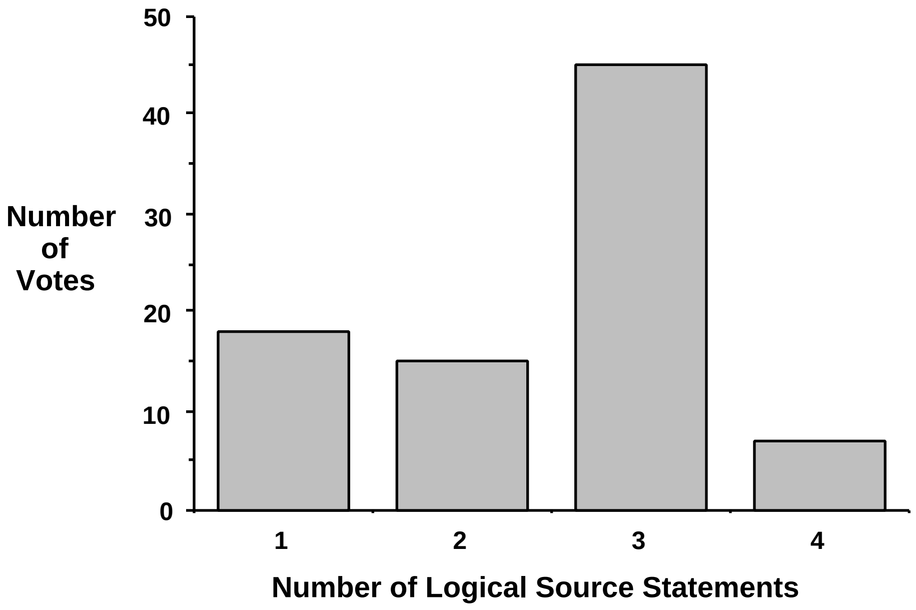


Figure 6-1 Poll Results—Logical Source Statements

Here is a slightly more complex example where people can disagree:

How would you prefer to count and classify the statements in this short C program that computes and prints a table for converting degrees Fahrenheit to degrees Celsius [Kernighan 88]?

```
#define    LOWER     0          /* lower limit of table */
#define    UPPER     300        /* upper limit */
#define    STEP      20         /* step size */

main()     /* print a Fahrenheit-Celsius conversion table */
{
    int fahr;
    for(fahr=LOWER; fahr<=UPPER; fahr=fahr+STEP)
        printf(“%4d %6.1f\n”, fahr, (5.0/9.0)*(fahr-32));
}
```

Will everyone agree with your results? How many different delimiters did you use as you were counting?

With these points settled, you may find it interesting to test your methods on this more exotic C language fragment:

```
for(;P("\n"),R--;P("|"))for(e=C;e--;P("_"+(*u++/8)%2))P("| "+(*u/4)%2);
```

At this point, we hope we have convinced you that there is more to counting logical source statements than many people realize. We also hope that we have not discouraged you from trying, as there certainly are benefits. We merely wish to point out that it is because logical statements are so difficult to define unambiguously that those who use them have a special responsibility to state explicitly and exactly the full set of rules they use to account for all possible events. Without full disclosure of all rules and delimiters, we simply cannot tell what information their measures contain.

For logical statement counts to be meaningful, users must state explicitly and completely the rules they employ. These requirements deserve special attention:

- Identifying the beginnings and endings of each statement type.
- Distinguishing among different statement types.
- Identifying and counting embedded statements, such as when one statement is used within or as an argument to another or is inserted or omitted between begin/end markers.
- Dealing with special situations, such as expression statements, modified comments, continuation lines, and the first or last statements in bodies or blocks of code.

## 6.3. Physical Source Lines

Physical source statements are source lines of code: one physical line equals one statement. The delimiter—or, more precisely, the terminator—for physical source statements is usually a special character or character pair such as *newline* or *carriage return--line feed*. When dealing with source code files, the particular delimiter used depends on the operating system. It is not a characteristic of the language.

Although how-to-count issues for physical source statements are easily settled, what-to-count issues bring out widely divergent views. As one example, we have shown the short C language program on the previous page to several audiences, and posed the question: "What do you think the SLOC count (physical source lines) for this program should be?" Responses varied even more widely than we anticipated, especially since one of our audiences was the COCOMO Users' Group, an assembly of practitioners who all use the same cost model and presumably the same definition for software size. Figure 6-2 shows the distribution of responses from two of our audiences. It includes the votes of the COCOMO people.

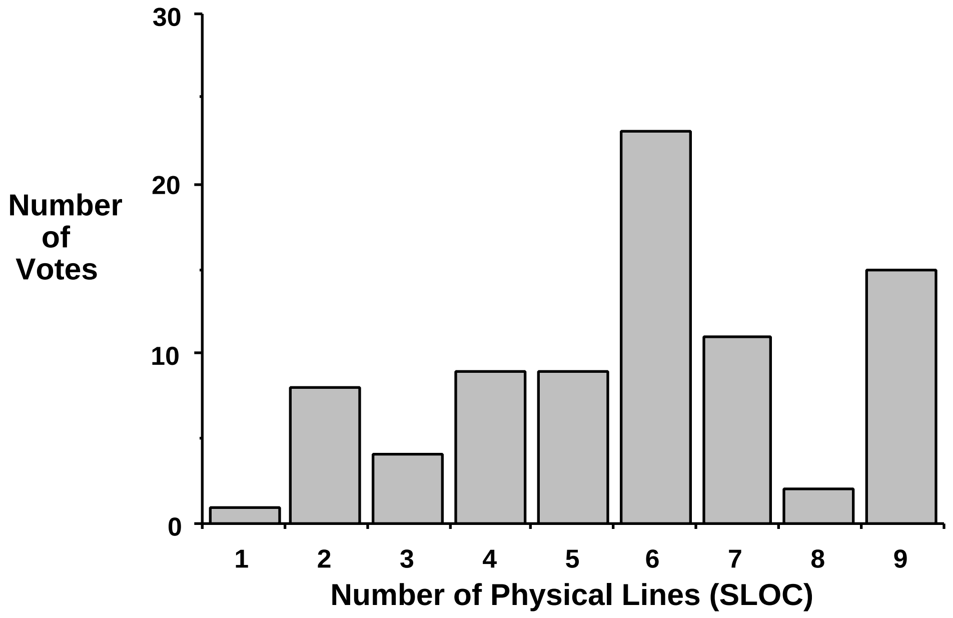


Figure 6-2 Poll Results—Physical Source Lines

Clearly, different perceptions exist with physical source lines just as with counts of logical source statements. There is an important difference though—these variations stem largely from different views as to *what* should be counted, not from any fundamental confusion or lack of definition for the counting unit.

Although what-to-count issues can be dealt with effectively with a simple checklist, physical source statements (i.e., lines of code) can be difficult to classify when more than one type of logical statement is on a physical line. (This was not a problem in the Fahrenheit-to-Celsius example.) Whenever programming languages permit multiple logical statements to exist on a single physical line, explicit rules for classifying lines according to statement types must be defined. The most straightforward way to do this seems to be through use of a precedence system. For this reason, we have included provisions for assigning classification precedences within the section of the checklist that addresses the **Statement type** attribute. At the risk of being overly directive, we have even gone so far as to impose our view of a precedence ordering. We hope that this makes use of the precedence facility more apparent.

Measures of physical source statements are most consistently obtained and statement types are most easily classified when source language modules are processed into standardized formats before counting. Pretty printers, formatters, and translators are tools that can provide these services. Standardized formatting with automated tools is a practice we recommend for all users of SLOC measures. Some pretty printers, however, have been programmed to introduce physical lines that organizations may not want to include in size counts. There is an argument in these cases for having a list of words that are treated as blank characters when counting nonblank lines. If this is done, then this list of words should be attached to the supplemental rules form.

Some people believe that counts of physical source statements can be misleadingly inflated through the use of lengthy names for variables and procedures. Others believe that the issue is clarity, not length; that descriptive names that add information make code more readable and easier to maintain; that this adds value; and that this gives reason for preferring physical counting over logical counting. These differing views can spark heated debate. Our conclusion is that the disagreements themselves give reason for measuring source code size in both ways. In fact, if measurements were available under consistent ground rules for the two different programming styles, then relating the results to development and maintenance costs might shed light on this and similar debates that in the past have been dominated more by emotion than evidence.

## 6.4. Physical and Logical Measures are Different Measures

The following Pascal fragment from a binary search tree provides some examples.

```
repeat
   if tree = nil
      then
         finished := true
      else
         with tree↑ do
            if key < data
               then
                  tree := left
               else if key > data
                  then
                     tree := right
                  else
                     finished := true
until finished;
```

How many logical statements are there? We think there are nine: one *repeat*, three *if*s, one *with*, and four assignments. Notice that in arriving at this count, we have implicitly treated *then* and *else* as keywords that function as statement delimiters. Since there is no requirement in Pascal that statements actually be present between two such delimiters (or between *begin…end* pairs, for that matter), you can begin to see some of the issues that those who count logical source statements must face when called upon to make their counting rules explicit and complete.

How many physical lines are there in the preceding example? Brute force would lead us to 15. But the code fragment could just as easily have been written this way:

```
repeat
   if tree = nil
      then finished := true
      else with tree↑ do
         if key < data
            then tree := left
            else if key > data
               then tree := right
               else finished := true
until finished;
```

Now there are but 10 physical lines. Note that the number of logical statements is unchanged. If your goal is to seek some sort of comparability between physical and logical source code measures, these issues merit attention.

Counts for physical and logical source statements should almost never be combined. Adding the two measures, except under very special conditions, almost guarantees meaningless results. The only instance that we know of where adding physical and logical statements has been advocated is in the recommendations for statement counting proposed for Ada COCOMO [Boehm 89]. Even here, the proposals follow very explicit rules, counting only physical lines for package specifications and only logical statements for package bodies. These proposals are motivated by the belief that logical statement counts based on terminal semicolons do not adequately account for the efforts required to produce many package specifications, and that some correction is needed to adequately relate the effects of the sizes of package specifications to development costs. They are no doubt motivated also by the fact that this method of counting maintains consistency with Ada's use of the term *statement* to refer only to executable statements.

To prevent confusion in reporting measures of size and in storing results in databases, counts of physical and logical source statements should always be maintained and reported separately. Anyone having an urge to combine the two kinds of measures can always do so after extracting the information from the database.

Combining two dissimilar measures is not prohibited, as long as those who do so recognize that combinations constitute models, not measures. The Ada COCOMO proposals are our view of one simple but practical model. This model says that the costs for one physical line in a package specification are approximately the same as for one logical statement in a package body. In time, and with consistent definitions in place, we would hope to be able to determine whether such equally weighted combinations of physical and logical counts provide the most effective inputs for Ada COCOMO, or whether some other weighting or model is even more effective.

## 6.5. Conclusion and Recommendation

The discussions here and in the preceding chapters lead us to conclude that physical source lines are both simpler to count and less subject to unstated assumptions than logical source statements. This will remain true until clear and exhaustive rules for identifying logical statement delimiters and statement types have been specified—a painstaking, language-dependent exercise. For these reasons, we recommend that organizations focus first on counts of physical source statements. Counts of logical source statements can be added later, if desired—but only after you have completed the requisite homework.

# 7. Reporting Other Rules and Practices— The Supplemental Forms

We have constructed three supplementary rules forms to provide structures for recording and describing language-dependent counting practices that cannot be dealt with adequately with a checklist. The first form is used to complete the definition of physical source statement counts. The second is used to complete the definition of rules used for identifying logical source statements. The third is used with both physical and logical source statements to explain the rules and practices used to identify dead code.

The kinds of rules these forms address include:

- Language-dependent rules for identifying the beginnings and endings of logical source statements.
- Language-dependent rules for identifying and distinguishing among different statement types.
- Environment-dependent tools and practices used to ensure that the dead code we do not want in size measures actually gets excluded.

Exact and explicit rules in the first two cases are especially important to those who specify, design, operate, or use results from automated source code counters. The third case is more an issue of verification or enforcement. Users of size data need to be able to judge for themselves whether adequate steps have been taken to ensure that reported counts are not inflated with unused, unproductive, and untested code. Assurances that dead code is excluded are unlikely to be trustworthy if the practices used for exclusion cannot be explained.

Figures 7-1, 7-2, and 7-3 show the supplemental forms we have created. Every definition checklist, when completed, should be accompanied by a completed rules form for each language to which the definition applies. Every set of measurement reports, when submitted, should also be accompanied by a completed form that explains the practices used to exclude dead code from reported results.

## 7.1. Physical Source Lines

| Rules for Counting Physical Source Lines | |
|---|---|
| For each source language to which the definition applies, provide the following information: **Language name**: | |
| Note: This information is required only for statement types that are excluded from counts or for which individual counts are recorded. | |
| **Executable lines**: List the rules used to identify executable lines. If special rules are used for constructs such as block statements, embedded statements, empty statements, or embedded comments, describe them. | **Comments:** List the rules used to identify beginnings and endings of comments. |
| **Declarations**: List the rules used to identify declaration lines. Explain how declarations are distinguished from executable statements. | **Modified comments**: If separate counts are made for modified lines, list the rules used to keep modifications to comments on lines with other code from being classified as modified statements of higher precedence. |
| **Compiler directives**: List the rules used to identify compiler directives. | **Special rules**: List any special rules that are used to classify the first or last statements of any sections of code. |

Figure 7-1 Rules Form—Counting Physical Source Lines

## 7.2. Logical Source Statements

| Rules for Counting Logical Source Statements | |
|---|---|
| For each source language to which this definition applies, provide the following information: **Language name**: | |
| **Executable statements**: List all rules and delimiters used to identify beginnings and endings of executable statements. If special rules are used for constructs such as block statements, embedded statements, empty statements, expression statements, or subprogram arguments, describe them. | **Comments:** If comments are counted, list the rules used to identify beginnings and endings of comment *statements*. Explain how, if at all, comment *statements* differ from physical source lines. |
| **Declarations**: List the rules and delimiters used to identify beginnings and endings of declarations. Explain how declarations are distinguished from executable statements. | **Special rules**: List any special rules or delimiters that are used to identify the first or last statements of any sections of code. |
| **Compiler directives**: List the rules and delimiters used to identify beginnings and endings of compiler directives. | **Exclusions:** List all keywords and symbols that, although set off by statement delimiters, are not counted as logical source statements. |

Figure 7-2 Rules Form—Counting Logical Source Statements

## 7.3. Dead Code

| Practices Used to Identify Inoperative Elements |
|---|
| List or explain the methods or rules used to identify:<br>**Intentionally bypassed statements and declarations**<br><br>**Unintentionally included dead code**<br>A. Unreachable, bypassed, or unreferenced elements (declarations, statements, or data stores) within modules:<br><br>B. Unused, unreferenced, or unaccessed modules or include files in code libraries:<br><br>C. Unused modules, procedures, or functions, linked into delivered products: |

Figure 7-3 Practices Used to Identify Inoperative Elements

# 8. Recording Measurement Results

Measurements, once made, must be entered into a database. Often the people who enter the results are not those who collect the original data. The results need somehow to get from those who collect them to those who put them into databases. To assist this process, we have constructed examples of the kinds of forms that you can use to record and transmit results of source statement counts (Figures 8-1 through 8-4). These examples are consistent with the definitions and data specifications we illustrated in Chapter 5, and they include information that tracks the data back to the definitions and to the entities measured.

Forms are very constraining instruments. Because they have flat surfaces, they are limited to displaying two dimensions at a time. This means that the only way we can use forms to record interactions of more than two dimensions is to hold all but two dimensions fixed, record the two dimensions of primary interest in an array, and then use additional forms or repetitions of the array to record the cases where other factors take on other values.

Each attribute on our checklist is a dimension. If paper is our medium, recording values for several attributes at once is cumbersome. There is, however, some good news. Most software modules will have relatively uniform characteristics in some of the dimensions. For example, either the entire module will be delivered or it will not. Either the entire module will be used within the primary product or it will not. Either the module will be master source code or it will be something else entirely. And often modules will be entirely of one language or of one origin. They will almost always, if small enough, have just one development status.

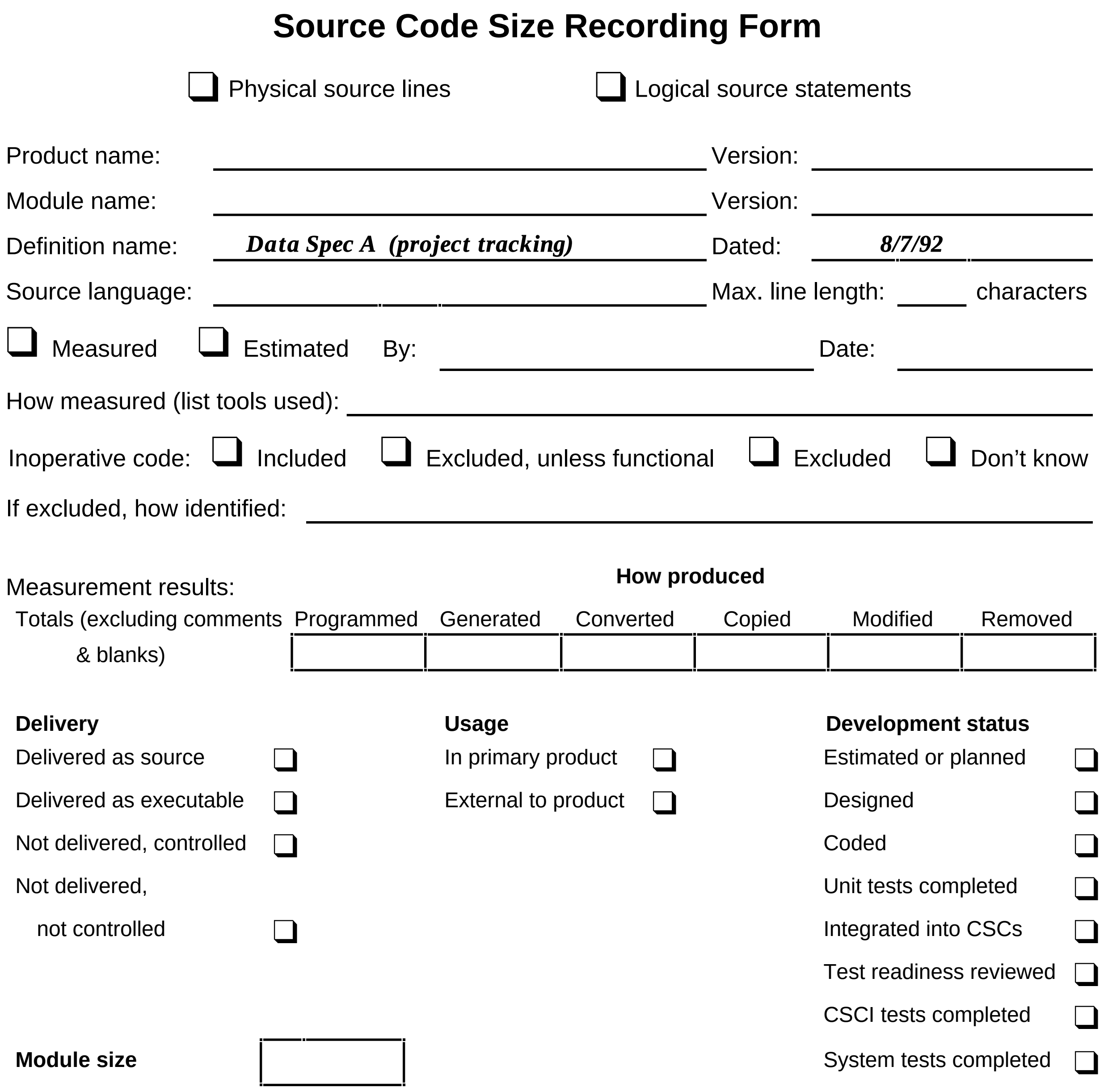
Our strategy in designing recording forms is to take advantage of these observations. The assumptions we make are that source code gets fed to source code counters in chunks, and that these chunks can be selected so that many characteristics (attributes) will have uniform values for the entire chunk. When these assumptions do not hold, we supplement our methods with special procedures. These include techniques such as dividing modules into smaller chunks, making separate passes through source statement counters for each programming language, and tagging individual statements or lines with codes (comments) to indicate different origins. We then use separate forms to record the results for each chunk, pass, or tagged set of statements.

In principle, one recording form should be used for each software entity measured. Thus, several or even many recording forms may be used for a given project or product. The primary purpose of the recording form is to get data entered correctly into the database.

We present these example forms not to say, “This is the way,” but merely to suggest the kinds of forms that can be helpful in ensuring that details requested by data specifications get recorded and entered correctly into the databases from which measurement reports will be generated. We encourage organizations to replace or alter these forms to meet their own needs. Copying is permitted, and Appendix E includes masters suitable for reproduction.

## 8.1. Example 1—Development Planning and Tracking

We have designed our first recording form (Figure 8-1) to support organizations that wish to use the data specification shown in Figure 5-4 (Data Spec A). This product tracking data specification has relatively few reporting requirements. We use our form to record and transmit low-level data. The form includes blocks to record the **Delivery** and **Usage** attributes because it costs little to do so and because recording values for these attributes may help to avoid misinterpretations. The form omits the **Origin** attribute since there is no need in Data Spec A to require all the code in each pass through a code counter to have the same origin.



**Source Code Size Recording Form**

☐ Physical source lines ☐ Logical source statements

Product name: ______________________ Version: __________

Module name: ______________________ Version: __________

Definition name: *Data Spec A (project tracking)* Dated: *8/7/92*

Source language: ______________________ Max. line length: _____ characters

☐ Measured ☐ Estimated By: ______________________ Date: __________

How measured (list tools used): ______________________

Inoperative code: ☐ Included ☐ Excluded, unless functional ☐ Excluded ☐ Don't know

If excluded, how identified: ______________________

Measurement results:

| | How produced | | | | | |
|---|---|---|---|---|---|---|
| Totals (excluding comments & blanks) | Programmed | Generated | Converted | Copied | Modified | Removed |
| | | | | | | |

| Delivery | | Usage | | Development status | |
|---|---|---|---|---|---|
| Delivered as source | ☐ | In primary product | ☐ | Estimated or planned | ☐ |
| Delivered as executable | ☐ | External to product | ☐ | Designed | ☐ |
| Not delivered, controlled | ☐ | | | Coded | ☐ |
| Not delivered, not controlled | ☐ | | | Unit tests completed | ☐ |
| | | | | Integrated into CSCs | ☐ |
| | | | | Test readiness reviewed | ☐ |
| | | | | CSCI tests completed | ☐ |
| **Module size** | | | | System tests completed | ☐ |

Figure 8-1 Recording Form for Data Spec A

## 8.2. Example 2—Planning and Estimating Future Products

We designed our second recording form (Figure 8-2) to support organizations that collect details such as Data Spec B (Figure 5-9) requests.

### Source Code Size Recording Form

☐ Physical source lines ☐ Logical source statements

Product name: ____________________ Version: ____________

Module name: ____________________ Version: ____________

Definition name: ***Data Spec B (project analysis)*** Dated: ***8/7/92***

Source language: ____________________ Max. line length: _____ characters

☐ Measured ☐ Estimated By: ____________________ Date: ____________

How measured (list tools used): ____________________

Inoperative code: ☐ Included ☐ Excluded, unless functional ☐ Excluded ☐ Don't know

If excluded, how identified: ____________________

Measurement results:

| | How produced | | | | | |
|---|---|---|---|---|---|---|
| **Statement type** | Programmed | Generated | Converted | Copied | Modified | Removed |
| Executable | | | | | | |
| Declaration | | | | | | |
| Compiler directive | | | | | | |
| Comment on own line | | | | | | |
| Comment on code line | | | | | | |

| **Delivery** | | **Usage** | | **Development status** | |
|---|---|---|---|---|---|
| Delivered as source | ☐ | In primary product | ☐ | Estimated or planned | ☐ |
| Delivered as executable | ☐ | External to product | ☐ | Designed | ☐ |
| Not delivered, controlled | ☐ | | | Coded | ☐ |
| Not delivered, | | | | Unit tests completed | ☐ |
| not controlled | ☐ | | | Integrated into CSCs | ☐ |
| | | | | Test readiness reviewed | ☐ |

|  |  |  |  |
|---|---|---|---|
|  |  | CSCI tests completed | ❑ |
| **Module size** |  | System tests completed | ❑ |

Figure 8-2 Recording Form for Data Spec B

## 8.3. Example 3—Reuse Tracking, Productivity Analysis, and Quality Normalization

The recording form in Figure 8-3 supports data specifications such as those illustrated in Figure 7-8 (Data Spec C).

# Source Code Size Recording Form

❑ Physical source lines ❑ Logical source statements

Product name: ______________________ Version: ____________

Module name: ______________________ Version: ____________

Definition name: *Data Spec C (reuse measurement)* Dated: *8/7/92*

Source language: ______________________ Max. line length: _____ characters

❑ Measured ❑ Estimated By: ______________________ Date: ____________

How measured (list tools used): ______________________

Inoperative code: ❑ Included ❑ Excluded, unless functional ❑ Excluded ❑ Don't know

If excluded, how identified: ______________________

Measurement results (executable statements plus declarations plus compiler directives):

**How produced**

| **Origin** | Programmed | Generated | Converted | Copied | Modified | Removed |
|---|---|---|---|---|---|---|
| New: no prior existence | | | | | | |
| Previous version, build, or release | | | | | | |
| Commercial, off-the-shelf software | | | | | | |
| Government furnished software | | | | | | |
| Another product | | | | | | |
| Local or modified lang. library or O/S | | | | | | |
| Other commercial library | | | | | | |
| Reuse library | | | | | | |
| Other component or library | | | | | | |

| **Delivery** | | **Usage** | | **Development status** | |
|---|---|---|---|---|---|
| Delivered as source | ❑ | In primary product | ❑ | Estimated or planned | ❑ |
| Delivered as executable | ❑ | External to product | ❑ | Designed | ❑ |
| Not delivered, controlled | ❑ | | | Coded | ❑ |
| Not delivered, not controlled | ❑ | | | Unit tests completed | ❑ |
| | | | | Integrated into CSCs | ❑ |
| | | | | Test readiness reviewed | ❑ |
| | | | | CSCI tests completed | ❑ |
| **Module size** | ☐ | noncomment, nonblank statements | | System tests completed | ❑ |

Figure 8-3 Recording Form for Data Spec C

## 8.4. Example 4—Support for Alternative Definitions and Recording Needs

Figure 8-4 shows a more detailed recording form that includes all attributes and values on the checklist except for the two **Origin** elements for unmodified vendor-supplied support software. You may be able to use this form to record interactions among **Statement types** and **How produced** in some situations where users ask for individual reports more detailed than those Chapter 5 illustrates. In other situations, you may need to design more specialized forms.

Standardized forms for reporting aggregated measures of multiple software units are even more difficult to devise due to the number of possible dimensions (attributes) that can be present at one time, each with multiple values. If you need data on the intersections of attributes other than **Statement type**, **How produced**, or **Origin**, it is probably best to construct individual forms that account for the multidimensional aspects of the attributes of interest. Chapter 9 discusses some ideas and examples for constructing forms for aggregating measurement results.

# Source Code Size Recording Form

☐ Physical source lines ☐ Logical source statements

| | | | |
|---|---|---|---|
| Product name: | ______ | Version: | ______ |
| Module name: | ______ | Version: | ______ |
| Definition name: | ______ | Dated: | ______ |
| Source language: | ______ | Max. line length: | ______ characters |

☐ Measured ☐ Estimated By: ______ Date: ______

How measured (list tools used):

Inoperative code: ☐ Included ☐ Excluded, unless functional ☐ Excluded ☐ Don't know

If excluded, how identified: ____________________

Measurement results:

| | How produced | | | | | |
|---|---|---|---|---|---|---|
| **Statement type** | Programmed | Generated | Converted | Copied | Modified | Removed |
| Executable | | | | | | |
| Declarations | | | | | | |
| Compiler directives | | | | | | |
| Comments | | | | | | |
| on their own lines | | | | | | |
| on lines with code | | | | | | |
| banners & spacers | | | | | | |
| blank comments | | | | | | |
| Blank lines | | | | | | |
| | | | | | | |

| **Origin** | |
|---|---|
| New: no prior existence | |
| Previous version, build, or release | |
| Commercial, off-the-shelf software | |
| Government furnished software | |
| Another product | |
| Vendor-supplied language library | |
| Vendor-supplied O/S (unmodified) | |
| Local or modified lang. library or O/S | |
| Other commercial library | |
| Reuse library | |
| Other component or library | |

**Delivery**

Delivered as source ☐

Delivered as executable ☐

Not delivered, controlled ☐

Not delivered, not controlled ☐

**Usage**

In primary product ☐

External to product ☐

**Development status**

Estimated or planned ☐

Designed ☐

Coded ☐

Unit tests completed ☐

Integrated into CSCs ☐

Test readiness reviewed ☐

CSCI tests completed ☐

System tests completed ☐

**Module size** ____

Figure 8-4 A Generalized Form for Recording Measurements of Source Code Size

# 9. Reporting and Requesting Measurement Results

After measures have been collected, they must be aggregated, summarized, and presented in ways that can be read and understood by the people who use them. When data is multi-dimensional, as it is when arrays of data elements are requested, the difficulties in presenting coherent summaries are greater than they seem at first glance.

The purpose of this chapter is to offer several forms of assistance:

- A reporting form that provides summary (grand) totals for individual elements in the checklist.
- An example of a form that you can use to summarize the project tracking data requested in Data Spec A.
- An outline of a process that users of measurement data can use to request the data they want reported.
- Two examples in which checklists are used to convey user requests for arrayed data.
- A form for requesting individual (marginal) summary reports.

Before presenting this assistance, we issue this very important caution: *You can't always get the total from the sum of the parts!* The reason is that intermediate totals may include statements drawn from code libraries and other modules. If these libraries or modules are used in separately counted bodies of code, double counting can occur. When computing total product size, you must always add only the most fundamental units you count; otherwise, some statements may get counted more than once.

## 9.1. Summarizing Totals for Individual Attributes

When the patterns in statement attributes are simple, you may be able to use recording forms like those in Chapter 8 to report summaries of measurement results. But when multiple values are recorded for several attributes, displays of aggregated results become complex. You will then need more specialized forms to summarize the data you collect.

The form we present in this section is useful for communicating a quick summary of "the big picture." It has the advantage that it tracks results directly back to the definition checklist, so that no associated definition or data specification is needed. The major disadvantage is that the form cannot show interactions among attributes—only the individual (i.e., marginal) totals formed by adding across the rows or down the columns of interacting cells. Because of this limitation, the form is seldom suitable for recording measurement results.

Figure 9-1 shows the kind of picture the first two pages of the data summary form can provide. Totals can also be reported for any of the individual language constructs on pages 3 though 5 of the checklist. The blank data summary in Appendix E includes these pages, should you wish to report data on the frequency of use of different language constructs.

## Data Summary—Source Code Size

Module ID *CSCI #3: Radar tracker v.1.0* Date counted: *8/7/92*

Language: *Ada* Reported by: *S. Smith*

**Measured as:**
[✓] **Physical source lines**
[ ] **Logical statements**

| | |
|---|---|
| **Counted** | *53,588* |
| **Estimated** | *6,750* |
| **Total** | *60,338* |

| | | **Totals include** | **Totals exclude** | **Individual totals** |
|---|---|---|---|---|
| **Statement type** | | | | |
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | | |
| 1 Executable **Order of precedence ->** | 1 | ✓ | | *33,772* |
| 2 Nonexecutable | | | | |
| 3 Declarations | 2 | ✓ | | *26,129* |
| 4 Compiler directives | 3 | ✓ | | *437* |
| 5 Comments | | | | |
| 6 On their own lines | 4 | | ✓ | |
| 7 On lines with source code | 5 | | ✓ | |
| 8 Banners and nonblank spacers | 6 | | ✓ | |
| 9 Blank (empty) comments | 7 | | ✓ | |
| 10 Blank lines | 8 | | ✓ | |
| 11 | | | | |
| 12 | | | | |
| **How produced** | | | | |
| 1 Programmed | | ✓ | | *39,188* |
| 2 Generated with source code generators | | ✓ | | *0* |
| 3 Converted with automated translators | | ✓ | | *0* |
| 4 Copied or reused without change | | ✓ | | *18,895* |
| 5 Modified | | ✓ | | *2,255* |
| 6 Removed | | | ✓ | |
| 7 | | | | |
| 8 | | | | |
| **Origin** | | | | |
| 1 New work: no prior existence | | ✓ | | *39,188* |
| 2 Prior work: taken or adapted from | | | | |
| 3 A previous version, build, or release | | ✓ | | *0* |
| 4 Commercial, off-the-shelf software (COTS), other than libraries | | ✓ | | *7,644* |
| 5 Government furnished software (GFS), other than reuse libraries | | ✓ | | *0* |
| 6 Another product | | ✓ | | *11,276* |
| 7 A vendor-supplied language support library (unmodified) | | | ✓ | |
| 8 A vendor-supplied operating system or utility (unmodified) | | | ✓ | |
| 9 A local or modified language support library or operating system | | ✓ | | *0* |
| 10 Other commercial library | | ✓ | | *0* |
| 11 A reuse library (software designed for reuse) | | ✓ | | *2,230* |
| 12 Other software component or library | | ✓ | | *0* |
| 13 | | | | |
| 14 | | | | |
| **Usage** | | | | |
| 1 In or as part of the primary product | | ✓ | | *60,338* |
| 2 External to or in support of the primary product | | | ✓ | |
| 3 | | | | |

Figure 9-1 Example Summary Size Report

| Module ID ***CSCI #3: Radar tracker v.1.0*** <br> Language: ***Ada*** | **Totals include** | **Totals exclude** | **Individual totals** |
|---|---|---|---|
| **Delivery** | | | |
| 1 Delivered | | | |
| 2 Delivered as source | ✔ | | ***58,108*** |
| 3 Delivered in compiled or executable form, but not as source | ✔ | | ***2,230*** |
| 4 Not delivered | | | |
| 5 Under configuration control | | ✔ | |
| 6 Not under configuration control | | ✔ | |
| 7 | | | |
| **Functionality** | | | |
| 1 Operative | ✔ | | ***60,114*** |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | | |
| 3 Functional (intentional dead code, reactivated for special purposes) | ✔ | | ***224*** |
| 4 Nonfunctional (unintentionally present) | | ✔ | ***unknown*** |
| 5 | | | |
| 6 | | | |
| **Replications** | | | |
| 1 Master source statements (originals) | ✔ | | — |
| 2 Physical replicates of master statements, stored in the master code | ✔ | | — |
| 3 Copies inserted, instantiated, or expanded when compiling or linking | | ✔ | |
| 4 Postproduction replicates—as in distributed, redundant, | | ✔ | |
| or reparameterized systems | | | |
| 5 | | | |
| **Development status** <br> *Each statement has one and only one status, usually that of its parent unit.* | | | |
| 1 Estimated or planned | | ✔ | |
| 2 Designed ***(estimated)*** | ✔ | | ***6,750*** |
| 3 Coded | ✔ | | ***8,374*** |
| 4 Unit tests completed | ✔ | | ***20,648*** |
| 5 Integrated into components | ✔ | | ***24,566*** |
| 6 Test readiness review completed | ✔ | | ***0*** |
| 7 Software (CSCI) tests completed | ✔ | | ***0*** |
| 8 System tests completed | ✔ | | ***0*** |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| **Language** <br> *List each source language on a separate line.* | | | |
| 1 | | | |
| 2 Job control languages | | ✔ | |
| 3 | | | |
| 4 Assembly languages | | ✔ | |
| 5 | | | |
| 6 Third generation languages ***Ada*** | ✔ | | ***60,338*** |
| 7 | | | |
| 8 Fourth generation languages | | ✔ | |
| 9 | | | |
| 10 Microcode | | ✔ | |
| 11 | | | |

Figure 9-1 Example Summary Size Report, Page 2

Although the data summary form bears strong resemblance to the definition checklist, there are some differences. The most noticeable is the third column on the right side, which provides a place to report the total counts for individual attribute values. The heading of the form differs from the definition checklist so that we can identify the software measured, the measurement date, the programming language, and the name of the individual preparing the report. Since the data summary form cannot show multidimensional interactions, the **Definition** and **Data array** boxes have been omitted. We have also added summary boxes at the top of the first page to record the total size and to show how much was estimated and how much was actually measured. These boxes help clarify summaries of counts made early in a product's life cycle, before actual code is available for measurement.

We have included a blank data summary form in Appendix E for those who would like a reproducible copy for local use. Pages 3 through 5 of the form are like those on the definition checklist, except that they include the third column. These pages permit users to clarify their counting rules, so that the definition checklist need not be attached. They also provide a place to summarize counts of individual programming constructs. This information can be used to track progress in adopting new languages and new software development paradigms, such as when organizations are just beginning to gain proficiency with Ada. It can also be used when developing methods and models for comparing size information across different programming languages.

When using the data summary form to report measurement results, you should attach supplementary rules forms for each source language used, just as you would with any other report of software size.

## 9.2. Summarizing Totals for Data Spec A (Project Tracking)

Figure 9-2 is a form we created for summarizing the tracking data we collect with Data Spec A. Results recorded on several copies of Figure 8-1 can be aggregated and reported with this form. A reproducible copy is included in Appendix E. Users should feel free to modify the form or to create alternatives that better meet their specialized needs.

### Source Code Size Reporting Form

❑ Physical source lines ❑ Logical source statements

Product or module name: ____________________ Version: ____________

Definition name: *Data Spec A (project tracking)* Dated: *7/9/92*

Source language: ____________________ Max. line length: _____ characters

❑ Measured ❑ Estimated By: ____________________ Date: __________

How measured (list tools used): ____________________

Inoperative code: ❑ Included ❑ Excluded, unless functional ❑ Excluded ❑ Don't know

If excluded, how identified: ____________________

| **Delivery** | | **Usage** | | **Total Removed** |
|---|---|---|---|---|
| Delivered | ❑ | In primary product | ❑ | |
| Not delivered | ❑ | External to product | ❑ | |

Measurement results:

| | **How produced** | | | | | |
|---|---|---|---|---|---|---|
| **Development Status** | Programmed | Generated | Converted | Copied | Modified | Total |
| Coded | | | | | | |
| Unit tests completed | | | | | | |
| Integrated into CSCs | | | | | | |
| Test readiness reviewed | | | | | | |
| CSCI tests completed | | | | | | |
| System tests completed | | | | | | |
| Total | | | | | | |

Figure 9-2 Summary Reporting Form for Data Spec A (Project Tracking)

## 9.3. A Structured Process for Requesting Summary Reports

When users of measurement data need insight into the intersections of different attributes, it helps to have methods for conveying their specialized requests to those who extract information from databases and prepare summary reports. This section outlines a structured process for conveying such requests. Once again the definition checklist plays a prominent role.

Figure 9-3 outlines the process. It begins with requesters using checklists to express their information needs. These personalized checklists define the data arrays and individual (marginal) totals the users want reported to them. The checklists can be either data array requests (as shown in Figures 9-4 and 9-5) or data summary requests (as shown in Figure 9-6). These information needs checklists then become the specifications for the reports prepared by the people who program or operate report generators.

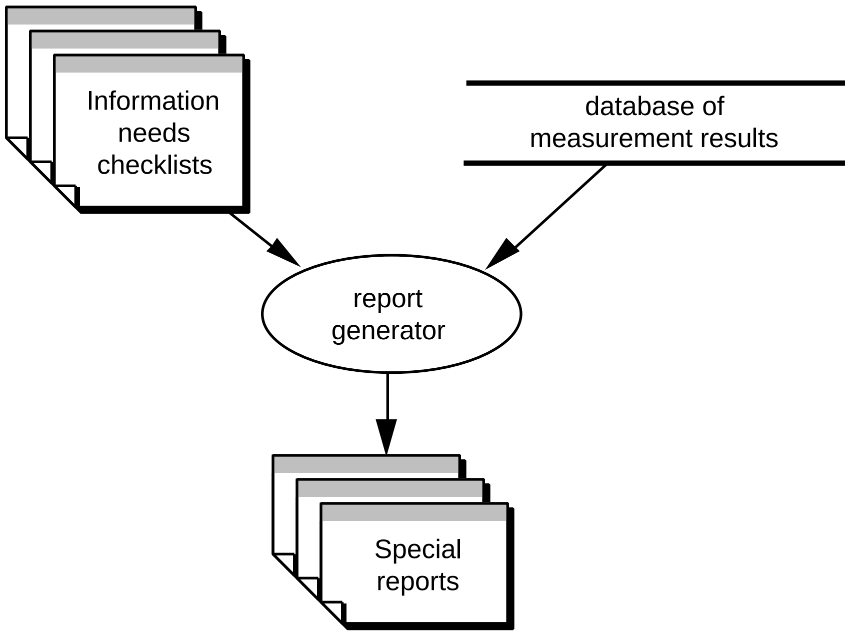


Figure 9-3 Using Definition Checklists to Specify Special Information Needs

## 9.4. Requesting Arrayed Data

Figures 9-4 and 9-5 are examples of the use of definition checklists to convey user requests for two-dimensional arrays. In each case, the rules for inclusion and exclusion that apply to attributes not in the arrays remain as illustrated in the basic definition in Chapter 5. As with data specifications, when the **Clarifications** pages of the definition checklist are attached, the requests stand independent of any definition of software size.

Figure 9-4 is a request for an array that shows the status of development for each method of production. It asks for a two-dimensional summary of the project tracking attributes whose measurement has already been specified by Data Spec A (Figure 5-4). Except for being a request from a user, it is identical to Data Spec A.

Figure 9-5, on the other hand, is not a duplicate of a prior measurement specification. It is a request for a subset of the project analysis data recorded by organizations that use Data Spec B (Figure 5-9).

When communicating requests for arrayed data to those who prepare reports, a useful rule is:

**Never ask for more than the number of attributes you can display at a time.**

Unless you are working with relatively sophisticated, computer-generated displays, this number is likely to be two. More dimensions can lead to confusion on the part of those who put your reports onto paper, and you may not get the information you want. Remember—you can always fill out as many different, two-dimensional, checklist-based requests as you want.

# Definition Checklist for Source Statement Counts

Definition name: ***Data array request— development status vs. production process***    Date: ***8/7/92***
Originator: ***S. Smith***

**Measurement unit:** **Physical source lines** ✓ | **Logical source statements** ☐

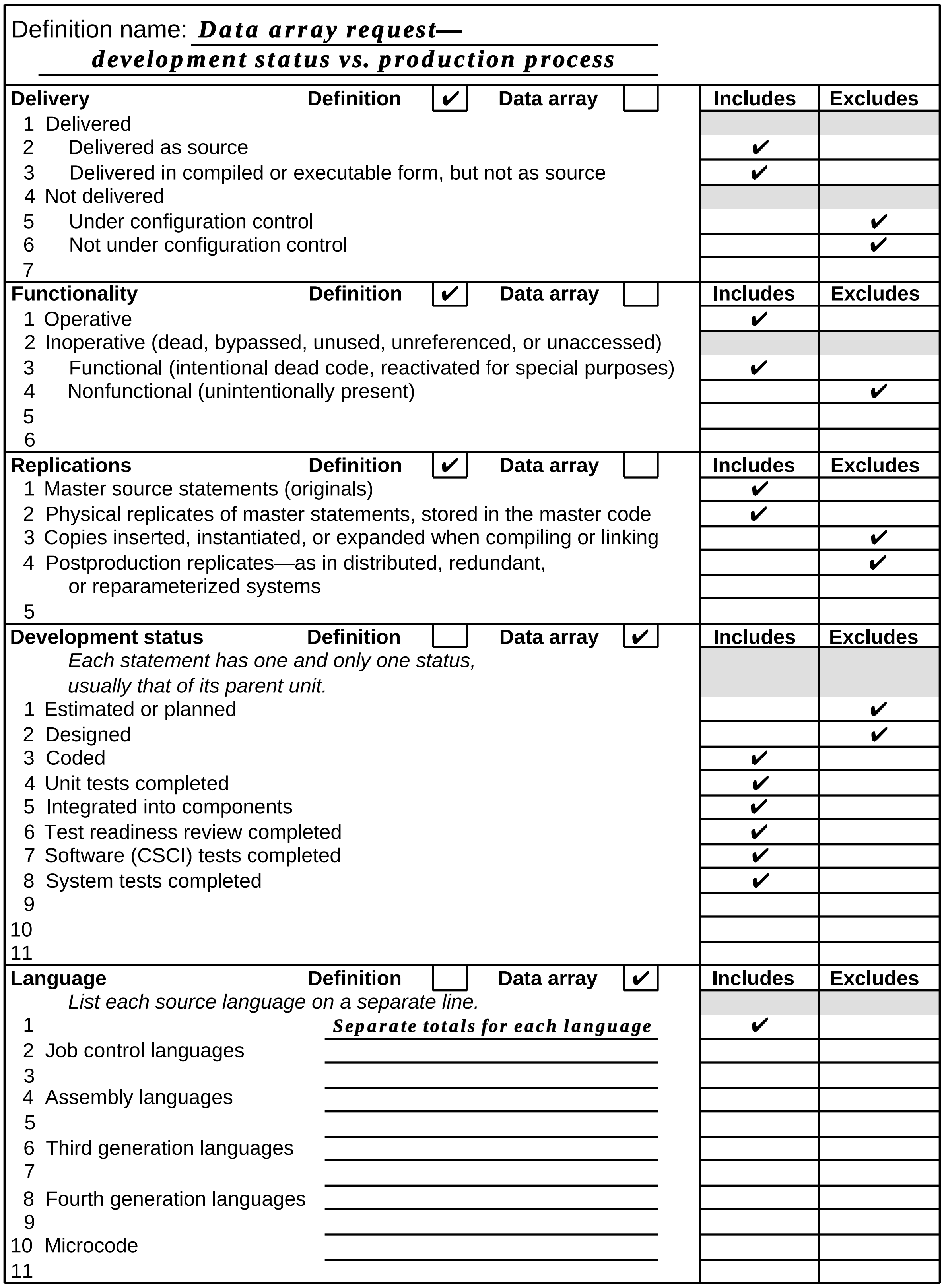
| **Statement type** Definition ✓ Data array ☐ | | **Includes** | **Excludes** |
|---|---|---|---|
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | ▨ | ▨ |
| 1 Executable **Order of precedence ->** | 1 | ✓ | |
| 2 Nonexecutable | ▨ | ▨ | ▨ |
| 3 Declarations | 2 | ✓ | |
| 4 Compiler directives | 3 | ✓ | |
| 5 Comments | ▨ | ▨ | ▨ |
| 6 On their own lines | 4 | | ✓ |
| 7 On lines with source code | 5 | | ✓ |
| 8 Banners and nonblank spacers | 6 | | ✓ |
| 9 Blank (empty) comments | 7 | | ✓ |
| 10 Blank lines | 8 | | ✓ |
| 11 | | | |
| 12 | | | |

| **How produced** Definition ☐ Data array ✓ | **Includes** | **Excludes** |
|---|---|---|
| 1 Programmed | ✓ | |
| 2 Generated with source code generators | ✓ | |
| 3 Converted with automated translators | ✓ | |
| 4 Copied or reused without change | ✓ | |
| 5 Modified | ✓ | |
| 6 Removed | ✓ | |
| 7 | | |
| 8 | | |

| **Origin** Definition ✓ Data array ☐ | **Includes** | **Excludes** |
|---|---|---|
| 1 New work: no prior existence | ✓ | |
| 2 Prior work: taken or adapted from | ▨ | ▨ |
| 3 A previous version, build, or release | ✓ | |
| 4 Commercial, off-the-shelf software (COTS), other than libraries | ✓ | |
| 5 Government furnished software (GFS), other than reuse libraries | ✓ | |
| 6 Another product | ✓ | |
| 7 A vendor-supplied language support library (unmodified) | | ✓ |
| 8 A vendor-supplied operating system or utility (unmodified) | | ✓ |
| 9 A local or modified language support library or operating system | ✓ | |
| 10 Other commercial library | ✓ | |
| 11 A reuse library (software designed for reuse) | ✓ | |
| 12 Other software component or library | ✓ | |
| 13 | | |
| 14 | | |

| **Usage** Definition ✓ Data array ☐ | **Includes** | **Excludes** |
|---|---|---|
| 1 In or as part of the primary product | ✓ | |
| 2 External to or in support of the primary product | | ✓ |
| 3 | | |

Figure 9-4 Request for a Report of Development Status vs. How Produced



Definition name: ***Data array request— development status vs. production process***

| **Delivery** — Definition [✓] Data array [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 Delivered | | |
| 2 Delivered as source | ✓ | |
| 3 Delivered in compiled or executable form, but not as source | ✓ | |
| 4 Not delivered | | |
| 5 Under configuration control | | ✓ |
| 6 Not under configuration control | | ✓ |
| 7 | | |

| **Functionality** — Definition [✓] Data array [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 Operative | ✓ | |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | |
| 3 Functional (intentional dead code, reactivated for special purposes) | ✓ | |
| 4 Nonfunctional (unintentionally present) | | ✓ |
| 5 | | |
| 6 | | |

| **Replications** — Definition [✓] Data array [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 Master source statements (originals) | ✓ | |
| 2 Physical replicates of master statements, stored in the master code | ✓ | |
| 3 Copies inserted, instantiated, or expanded when compiling or linking | | ✓ |
| 4 Postproduction replicates—as in distributed, redundant, or reparameterized systems | | ✓ |
| 5 | | |

| **Development status** — Definition [ ] Data array [✓] | **Includes** | **Excludes** |
|---|---|---|
| *Each statement has one and only one status, usually that of its parent unit.* | | |
| 1 Estimated or planned | | ✓ |
| 2 Designed | | ✓ |
| 3 Coded | ✓ | |
| 4 Unit tests completed | ✓ | |
| 5 Integrated into components | ✓ | |
| 6 Test readiness review completed | ✓ | |
| 7 Software (CSCI) tests completed | ✓ | |
| 8 System tests completed | ✓ | |
| 9 | | |
| 10 | | |
| 11 | | |

| **Language** — Definition [ ] Data array [✓] | **Includes** | **Excludes** |
|---|---|---|
| *List each source language on a separate line.* | | |
| 1 ***Separate totals for each language*** | ✓ | |
| 2 Job control languages | | |
| 3 | | |
| 4 Assembly languages | | |
| 5 | | |
| 6 Third generation languages | | |
| 7 | | |
| 8 Fourth generation languages | | |
| 9 | | |
| 10 Microcode | | |
| 11 | | |

Figure 9-4 Request for a Report of Development Status vs. How Produced, Page 2

## Definition Checklist for Source Statement Counts

Definition name: ***Data array request— statement type vs. programming language***

Date: ***8/7/92***

Originator: ***J. Jones***

**Measurement unit:** **Physical source lines** [✓] **Logical source statements** [ ]

| **Statement type** — **Definition** [ ] **Data array** [✓] | | **Includes** | **Excludes** |
|---|---|---|---|
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | |
| 1 Executable — **Order of precedence ->** | 1 | ✓ | |
| 2 Nonexecutable | | | |
| 3 Declarations | 2 | ✓ | |
| 4 Compiler directives | 3 | ✓ | |
| 5 Comments | | | |
| 6 On their own lines | 4 | | ✓ |
| 7 On lines with source code | 5 | | ✓ |
| 8 Banners and nonblank spacers | 6 | | ✓ |
| 9 Blank (empty) comments | 7 | | ✓ |
| 10 Blank lines | 8 | | ✓ |
| 11 ***Textual comments: #6 + #7, even if on lines with other*** | — | ✓ | |
| 12 ***statements*** | | | |

| **How produced** — **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 Programmed | ✓ | |
| 2 Generated with source code generators | ✓ | |
| 3 Converted with automated translators | ✓ | |
| 4 Copied or reused without change | ✓ | |
| 5 Modified | ✓ | |
| 6 Removed | ✓ | |
| 7 | | |
| 8 | | |

| **Origin** — **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 New work: no prior existence | ✓ | |
| 2 Prior work: taken or adapted from | | |
| 3 A previous version, build, or release | ✓ | |
| 4 Commercial, off-the-shelf software (COTS), other than libraries | ✓ | |
| 5 Government furnished software (GFS), other than reuse libraries | ✓ | |
| 6 Another product | ✓ | |
| 7 A vendor-supplied language support library (unmodified) | | ✓ |
| 8 A vendor-supplied operating system or utility (unmodified) | | ✓ |
| 9 A local or modified language support library or operating system | ✓ | |
| 10 Other commercial library | ✓ | |
| 11 A reuse library (software designed for reuse) | ✓ | |
| 12 Other software component or library | ✓ | |
| 13 | | |
| 14 | | |

| **Usage** — **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 In or as part of the primary product | ✓ | |
| 2 External to or in support of the primary product | | ✓ |
| 3 | | |

Figure 9-5 Request for a Report of Statement Type vs. Programming Language

Definition name: ***Data array request—***
***statement type vs. programming language***

| **Delivery** | **Definition** ✓ | **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|---|
| 1 Delivered | | | ▨ | ▨ |
| 2 Delivered as source | | | ✓ | |
| 3 Delivered in compiled or executable form, but not as source | | | ✓ | |
| 4 Not delivered | | | ▨ | ▨ |
| 5 Under configuration control | | | | ✓ |
| 6 Not under configuration control | | | | ✓ |
| 7 | | | | |

| **Functionality** | **Definition** ✓ | **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|---|
| 1 Operative | | | ✓ | |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | | ▨ | ▨ |
| 3 Functional (intentional dead code, reactivated for special purposes) | | | ✓ | |
| 4 Nonfunctional (unintentionally present) | | | | ✓ |
| 5 | | | | |
| 6 | | | | |

| **Replications** | **Definition** ✓ | **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|---|
| 1 Master source statements (originals) | | | ✓ | |
| 2 Physical replicates of master statements, stored in the master code | | | ✓ | |
| 3 Copies inserted, instantiated, or expanded when compiling or linking | | | | ✓ |
| 4 Postproduction replicates—as in distributed, redundant, or reparameterized systems | | | | ✓ |
| 5 | | | | |

| **Development status** | **Definition** ✓ | **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|---|
| *Each statement has one and only one status, usually that of its parent unit.* | | | ▨ | ▨ |
| 1 Estimated or planned | | | | ✓ |
| 2 Designed | | | | ✓ |
| 3 Coded | | | ✓ | |
| 4 Unit tests completed | | | ✓ | |
| 5 Integrated into components | | | ✓ | |
| 6 Test readiness review completed | | | ✓ | |
| 7 Software (CSCI) tests completed | | | ✓ | |
| 8 System tests completed | | | ✓ | |
| 9 | | | | |
| 10 | | | | |
| 11 | | | | |

| **Language** | **Definition** ☐ | **Data array** ✓ | **Includes** | **Excludes** |
|---|---|---|---|---|
| *List each source language on a separate line.* | | | ▨ | ▨ |
| 1 | ***Separate totals for each language*** | | ✓ | |
| 2 Job control languages | | | | |
| 3 | | | | |
| 4 Assembly languages | | | | |
| 5 | | | | |
| 6 Third generation languages | | | | |
| 7 | | | | |
| 8 Fourth generation languages | | | | |
| 9 | | | | |
| 10 Microcode | | | | |
| 11 | | | | |

Figure 9-5 Request for a Report of Statement Type vs. Programming Language, Page 2

## 9.5. Requesting Individual (Marginal) Totals

Figures 9-4 and 9-5 ask for arrayed data. Figure 9-6, on the other hand, is an example of a request for simple marginal totals like those illustrated in Figure 9-1. Checks in the third column of this form request individual totals.

The full form with the entries left blank is presented in Appendix E. This form can be used by itself to ask for simple summaries, or it can be used in conjunction with requests for arrayed data to get the marginal totals that go with the arrays. It can also be used as part of a data specification to ask for counting and recording of selected attribute values that do not need to be arrayed against other attributes.

Pages 3 through 5 of the data request form in Appendix E can be used not only to state the rules you want applied when counting individual language constructs, but also to request frequency counts for construct use. As with other summary totals, checks in the right-hand column ask for these details to be collected and reported without disturbing or modifying the basic definition for size or any other data request.

## Data Summary Request—Source Code Size

Definition name: *Standard SLOC* Date: *8/7/92*

*Data Spec A — marginal totals* Requester *H. Burger*

| **Measured as:** [✓] **Physical source lines** [ ] **Logical statements** | | **Totals include** | **Totals exclude** | **Individual totals** |
|---|---|---|---|---|
| **Statement type** *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | | |
| 1 Executable **Order of precedence ->** | 1 | ✓ | | |
| 2 Nonexecutable | | | | |
| 3 Declarations | 2 | ✓ | | |
| 4 Compiler directives | 3 | ✓ | | |
| 5 Comments | | | | |
| 6 On their own lines | 4 | | ✓ | |
| 7 On lines with source code | 5 | | ✓ | |
| 8 Banners and nonblank spacers | 6 | | ✓ | |
| 9 Blank (empty) comments | 7 | | ✓ | |
| 10 Blank lines | 8 | | ✓ | |
| 11 | | | | |
| 12 | | | | |
| **How produced** | | | | |
| 1 Programmed | | ✓ | | ✓ |
| 2 Generated with source code generators | | ✓ | | ✓ |
| 3 Converted with automated translators | | ✓ | | ✓ |
| 4 Copied or reused without change | | ✓ | | ✓ |
| 5 Modified | | ✓ | | ✓ |
| 6 Removed | | | ✓ | ✓ |
| 7 | | | | |
| 8 | | | | |
| **Origin** | | | | |
| 1 New work: no prior existence | | ✓ | | |
| 2 Prior work: taken or adapted from | | | | |
| 3 A previous version, build, or release | | ✓ | | |
| 4 Commercial, off-the-shelf software (COTS), other than libraries | | ✓ | | |
| 5 Government furnished software (GFS), other than reuse libraries | | ✓ | | |
| 6 Another product | | ✓ | | |
| 7 A vendor-supplied language support library (unmodified) | | | ✓ | |
| 8 A vendor-supplied operating system or utility (unmodified) | | | ✓ | |
| 9 A local or modified language support library or operating systen | | ✓ | | |
| 10 Other commercial library | | ✓ | | |
| 11 A reuse library (software designed for reuse) | | ✓ | | |
| 12 Other software component or library | | ✓ | | |
| 13 | | | | |
| 14 | | | | |
| **Usage** | | | | |
| 1 In or as part of the primary product | | ✓ | | |
| 2 External to or in support of the primary product | | | ✓ | |
| 3 | | | | |

Figure 9-6 Request Specification for a Summary Report of Attribute Totals

| Definition name: ***Standard SLOC*** ***Data Spec A — marginal totals*** | **Totals include** | **Totals exclude** | **Individual totals** |
|---|---|---|---|
| **Delivery** | | | |
| 1 Delivered | | | |
| 2 Delivered as source | ✔ | | |
| 3 Delivered in compiled or executable form, but not as source | ✔ | | |
| 4 Not delivered | | | |
| 5 Under configuration control | | ✔ | |
| 6 Not under configuration control | | ✔ | |
| 7 | | | |
| **Functionality** | | | |
| 1 Operative | ✔ | | |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | | |
| 3 Functional (intentional dead code, reactivated for special purposes) | ✔ | | |
| 4 Nonfunctional (unintentionally present) | | ✔ | |
| 5 | | | |
| 6 | | | |
| **Replications** | | | |
| 1 Master source statements (originals) | ✔ | | |
| 2 Physical replicates of master statements, stored in the master code | ✔ | | |
| 3 Copies inserted, instantiated, or expanded when compiling or linking | | ✔ | |
| 4 Postproduction replicates—as in distributed, redundant, or reparameterized systems | | ✔ | |
| 5 | | | |
| **Development status** *Each statement has one and only one status, usually that of its parent unit.* | | | |
| 1 Estimated or planned | | ✔ | |
| 2 Designed | | ✔ | |
| 3 Coded | ✔ | | ✔ |
| 4 Unit tests completed | ✔ | | ✔ |
| 5 Integrated into components | ✔ | | ✔ |
| 6 Test readiness review completed | ✔ | | ✔ |
| 7 Software (CSCI) tests completed | ✔ | | ✔ |
| 8 System tests completed | ✔ | | ✔ |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| **Language** *List each source language on a separate line.* | | | |
| 1 ***Separate totals for each language*** | ✔ | | ✔ |
| 2 Job control languages | | | |
| 3 | | | |
| 4 Assembly languages | | | |
| 5 | | | |
| 6 Third generation languages | | | |
| 7 | | | |
| 8 Fourth generation languages | | | |
| 9 | | | |
| 10 Microcode | | | |
| 11 | | | |

Figure 9-6 Request Specification for a Summary Report of Attribute Totals, Page 2

# 10. Meeting the Needs of Different Users

As demonstrated in Chapter 9, one of the useful features of checklists with independent attributes and mutually exclusive classes is that you can use them to request alternative reports for different purposes. This means that you can use them to satisfy the needs of different users. For example, project managers, configuration controllers, cost estimators, reuse trackers, and quality improvers often need different measurements or different levels of detail to do their jobs. With the checklist, each can describe his or her own data requirements. You can then use additional checklists to consolidate these requests into a common definition and its supporting data specifications. This provides a way to resolve seemingly conflicting views. Figure 10-1 outlines the process we have in mind.

Users will often need to negotiate and compromise during consolidation as their proposals get balanced against costs of collecting data and requirements for internal consistency. The fact that each user has identified the coverage he or she needs in a succinct, structured way greatly aids negotiation.

We recommend settling on a common definition for size before preparing data specifications. Once the basic coverage is defined, designing specifications for additional data elements is relatively straightforward. Moreover, the ability that the checklist gives to designate individual attribute values for measurement provides a safety valve that facilitates compromise on elements included in a basic definition. When users can get the data they want through reports on individual (possibly excluded) data elements, they are often willing to accept the coverage rules that others want in their common definition.

Preparing consolidated data specifications consists of checking off, in the inclusion columns, all elements for which individual reports are requested. You can almost always do this without changing the basic size definition, since a data specification is nothing more than a request for counting other elements in addition to the agreed upon measure for size. When the consolidations are complete, the inclusion columns of the data specifications become specifications for (1) fields to be included in the database and (2) for forms for data collection.

When you measure, you will use the inclusion and exclusion rules in the consolidated definition to count and record values for the common measure for size. You will also count and record individual data elements according to the rules in their respective data specifications. You will then enter each of these observations into your measurement database. Finally, you will generate individualized reports by aggregating data from the database according to the instructions given in each user needs checklists. By following this process, different groups of users can all have their own customized size reports, without requiring repeated measurement efforts.

The steps we recommend for constructing and using size definitions are listed in Figure 10-2. As we have said, we view definition to be an active process that requires both negotiation and communication. The definition checklist provides a structure for making these activities easier than they otherwise would be.

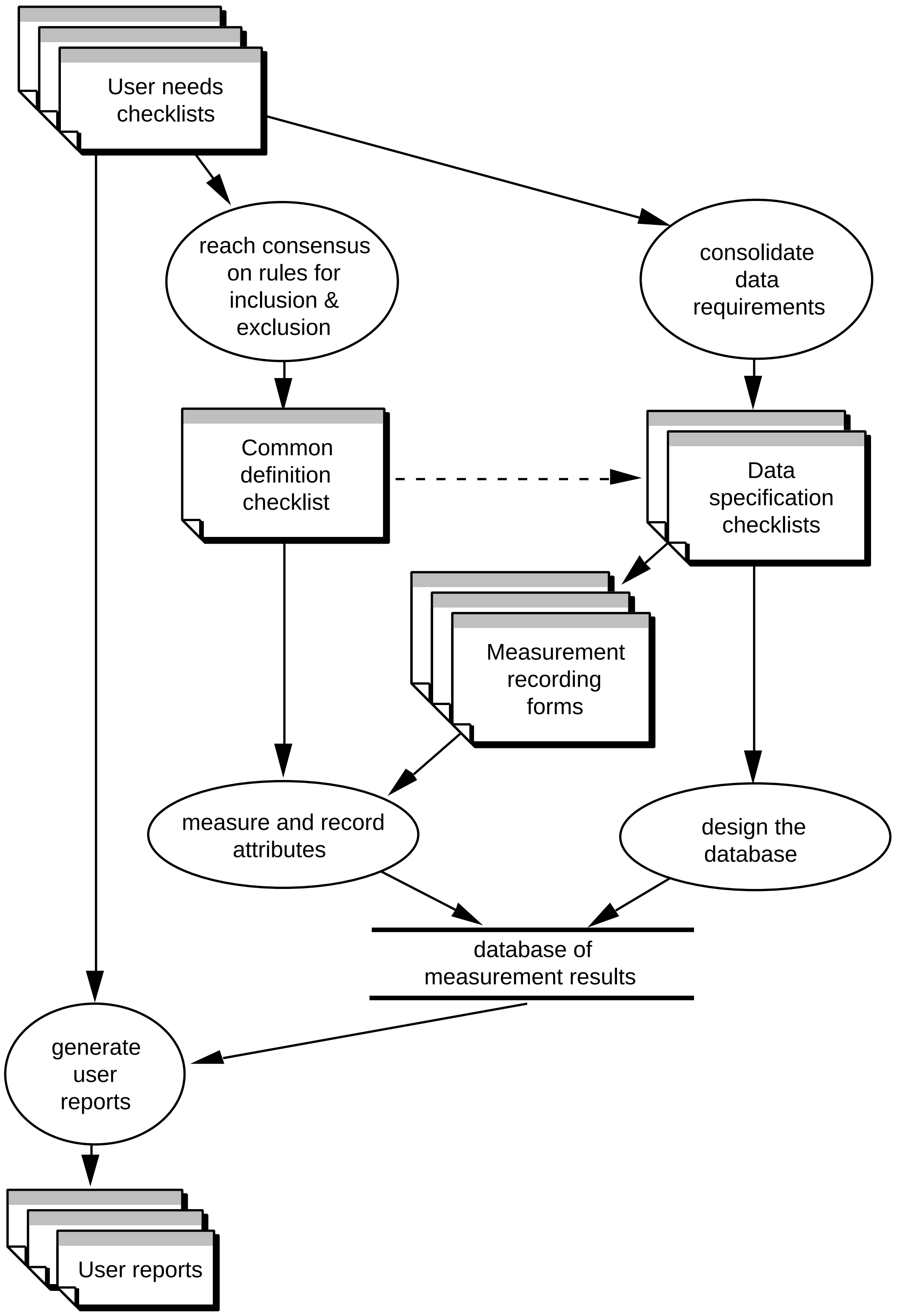

User needs checklists
reach consensus on rules for inclusion & exclusion
consolidate data requirements
Common definition checklist
Data specification checklists
Measurement recording forms
measure and record attributes
design the database
database of measurement results
generate user reports
User reports

Figure 10-1 Use of Checklists to Construct Common Definitions While Meeting the Needs of Different Measurement Users

**Steps for Constructing and Using Definitions for Source Code Size**

1. List the reasons why your organization wants measures for source code size. Who will use your measurement results? How? What decisions will be based on these reports?
2. Distribute copies of the definition and data summary request checklists to the groups who will use the measurement results. Have these groups use the checklists to state the information that they need to perform their jobs. Have them also state any special rules that they want applied during measurement. Use the clarification sections of the definition checklist and the supporting rules forms to aid this process.
3. Consolidate the inclusion and exclusion requirements of the different users into a single common definition for size. Record the results in a single checklist.
4. Consolidate the data array requirements of the different users into data specifications and record the results in separate copies of the definition checklist. Record any additional requirements for grand totals or marginal distributions on consolidated data summary requests.
5. Use the results of step 4 as design requirements for your measurement database.
6. Use the users' proposed clarifications and supporting rules forms to consolidate and reach consensus on the special instructions to be followed when recording and reporting measurement results. Attach the consolidated rules to the common definition checklist.
7. With the results of steps 3, 4, and 6 as guides, prepare code counters and/or forms for collecting and recording measurement results.
8. Collect measurements and enter the results in your database.
9. Using your users' data requests as specifications, generate the reports they need.
10. Attach copies of the definition checklist, supporting rules forms, and data specifications to each set of measurement reports.

Figure 10-2 Steps for Constructing and Using Definitions for Source Code Size

As the structured process described in Figures 10-1 and 10-2 shows, the source statement checklist can be used in different but nonconflicting ways to help you perform any of these functions:

- Create a general-purpose, standardized definition for size.
- Identify and specify the data elements needed for different management purposes. Examples of different purposes include project tracking, cost estimating, quality normalization, reuse management, and process improvement.
- Adjust the granularity and quantity of reported detail as organizations progress to higher levels of process maturity, all without changing previously agreed upon definitions.
- Identify data fields required for size databases.
- Specify the content of specialized reports to be prepared for individual users.

With a definition checklist like the one in Figure 3-2 and a structured process like the one in Figures 10-1 and 10-2, users can say exactly what data they want collected and recorded, so that it can be aggregated to form both generic and specialized measures of software size. The result is that a single, combined definition together with consolidated specifications for collecting and recording measurement results can be made to serve the management needs of different users.

# 11. Applying Size Definitions to Builds, Versions, and Releases

We measure size to help us plan, control, and improve activities that go into producing and maintaining software systems. When we think of size in the context of activities, it is apparent that at least two views are valid: size as a snapshot of product status and size as a measure of the amount produced (or consumed) between two points in the process.

Understanding the distinctions between these views is important when applying size measures to multiple builds, versions, and releases. Whenever physical source lines or logical source statements are used in these instances, activity sizes cannot be obtained simply by subtracting starting static size values from ending static size values—the difference between two static measures is inadequate for describing the size of the tasks performed. Instead, we require methods that permit us to describe size in ways that can be mapped more directly to effort.

Checklist-based definitions like the one in Chapter 5 provide the kind of framework we need, but we must use care lest we get tangled in terminology. For example, when discussing a second build, it is not immediately clear just what terms like *copied*, *modified*, and *removed* may mean. Nor is it immediately clear just how pronouncements like “Build 2 includes 50,000 source lines of commercial software” should be interpreted, especially when that software has already been included in the description of Build 1. In either case, we can easily confuse elements that are legitimately part of the work of Build 2 with those that are accounted for in Build 1.

This seems to be an instance where a picture describes the solution more adequately than words. Figure 11-1 shows how we apply the definition of Chapter 5 to multiple builds. We use the same rules for multiple versions and multiple releases.

Figure 11-1 shows the first two builds of a multistep development. The origins of pre-existing statements used in the first step (Build 1) are labeled Origin 1. The origins of pre-existing statements used in the second step are labeled Origin 2. Note that all statements from Build 1 become classified as coming from a prior build when describing the size of Build 2. This includes statements programmed and generated in Build 1. If you wish to track statements from their previous origins, you have two options—you can go back and look at the data for Build 1, or you can lump Builds 1 and 2 together and talk about the total size of the combined activity that begins with the start of development and runs through the completion of Build 2.

Because it is easy to misinterpret counting rules for the **How produced** and **Origin** attributes when measuring sizes for sequential builds, versions, or releases, we recommend attaching an annotated version of Figure 11-1 or a similar figure to the definition checklist and data specifications whenever measures of source code size are used for multistep developments.

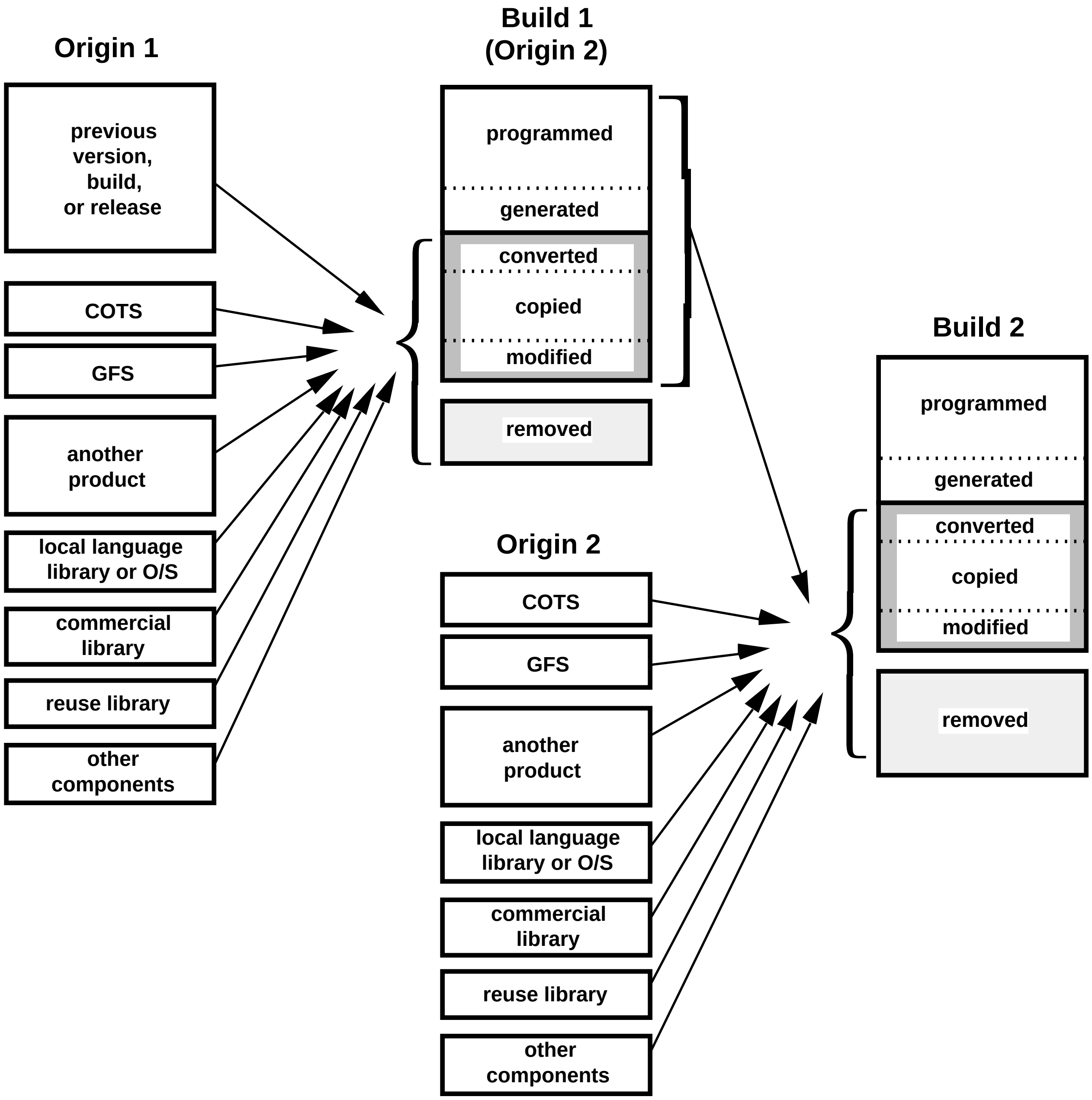


Figure 11-1 Applying Size Definitions to Builds, Versions, and Releases

# 12. Recommendations for Initiating Size Measurement

This chapter collects and summarizes some of the more important conclusions we reached while preparing this report. We present these conclusions as recommendations. Many of them have been discussed in earlier chapters.

## 12.1. Start with Physical Source Lines

Our principal recommendation is that organizations adopt physical source lines of code (SLOC) as one of their first measures of software size. Subsequently, if they wish, they can add counts of logical source statements or even other size measures, but only after the requisite language-specific rules have been defined and automated counters have been built.

Our preference for starting with physical source lines rather than logical source statements (or other measures) is not entirely arbitrary. Several reasons have been discussed already in Chapters 4 and 6, and others are given in Appendix C. Our conclusion is somewhat different than the one reached in the IEEE proposed *Standard for Software Productivity Metrics* [IEEE 92]. That draft standard expresses a preference for logical source statements. We acknowledge the theoretical advantages that counts of logical source statements offer in terms of relative independence from formatting style, but a number of observations persuade us to believe that counting source lines of code encounters fewer difficulties. Some of these observations are as follows:

- It is easier and cheaper to build automated counters for physical lines than it is for logical statements. In fact, if you simply count all nonblank lines, you need know nothing about the source language at all. Alternatively, if you want to skip comment lines so that counts for noncomment, nonblank source statements are obtained, only minor tailoring for individual languages is needed. The definition for SLOC proposed in Figure 5-1 is just such a definition.
- Rules for determining when logical source statements begin and end are complex and different for every source language. Different kinds of statements often have different kinds of delimiters, and methods for identifying and treating embedded statements and expression statements must be spelled out. Some logical source "statements" may even be undefined, as when certain elements such as declarations and comments are not classified as statements by the respective language standards or reference manuals. Physical lines, on the other hand, have consistent delimiters, regardless of language.
- It is generally easier for most people to interpret and compare measures of physical line counts, especially in environments where coding standards, code formatters, or pretty printers are used.
- Pressures to revert to overly simplistic counting rules for logical source statements are high. These shortcuts lead easily to measures that are not comparable across

different programming languages. For example, popular rules such as “count terminal semicolons” do just that—they produce counts of semicolons, not counts of logical source statements. Different languages use semicolons in entirely different ways, making consistent interpretations difficult.

- Most historical data is in terms of physical source lines. Organizations should not be forced to abandon these historical references.

Although we recommend starting with physical source lines as one of the first measures of software size, we do not suggest that anyone abandon any measure that is now being used to good effect. Nor do we suggest that counts of physical source lines are the best measure for software size. It is unlikely that they are. But until we get firm, well-defined evidence that other measures are better, we opt for simplicity—in particular, for the simplest set of explicit rules that can be applied across many languages.

Some advocates of function point measures may disagree strongly with our primary recommendation. This is well and good, and we encourage all function point users to continue to expand upon their practices. In fact, in some environments function point measures appear to have significant advantages over source statement counts. For example, they are language-independent, often solution-independent, and usually computable early in development life cycles, even before specific product designs are available.

But function point measures have three properties that make us hesitate to recommend them as something that every organization should implement:

1. Function points do not support project tracking. Because function points are not contained within units, it is very difficult during development to say what percentage of the total function point count is coded, what percentage is unit tested, what percentage is integrated, etc.
2. Automated function point counters do not yet exist. To the best of our knowledge, no-one today has the ability to feed source code or any other completed software artifact to an automated tool that computes function points.
3. Function points are not equally applicable to all kinds of software. For example, few if any organizations have reported consistent success in computing valid function point counts for things like embedded, real-time systems, continuously operating systems, or systems that spend large amounts of computational effort to generate a small number of numerical results.

These, then, are our reasons for recommending that most organizations adopt physical source lines as one of their first measures of software size. Once counts of physical source lines are successfully implemented, one of the next measures we would look to would be counts of computer software units (CSUs), as defined in DOD-STD-2167A. We see no reason why checklists much like the ones in this report could not be used to clarify and characterize counts of software units.

## 12.2. Use a Checklist-Based Process to Define, Record, and Report Your Measurement Results

1. When starting with physical source lines of code (SLOC), adopt the definition illustrated in Figure 5-1. Our abilities to compare projects and learn from history will be helped if we all speak a common language. (Alternatively, if you use logical source statements, consider adopting the definition in Figure 5-2.)
2. If you need an alternative definition of size for internal work, use the processes in Figures 10-1 and 10-2 to construct one that meets the needs of the principal users of your measurement results.
3. Use the processes in Chapters 9 and 10 to identify the additional data elements that you wish to have recorded. Be careful not to ask for data on too many attributes, especially when first implementing your basic definition. Data specifications like those in Figures 5-4 (Data Spec A—Project Tracking) and 5-9 (Data Spec B—Project Analysis), or subsets of them, are appropriate starting points for many organizations.
4. Review and augment, as needed, the language-specific clarifications on pages 3 through 5 of the definition checklist. Ensure that *all* potentially confusing peculiarities of *all* languages you use are identified, and that rules are assigned for counting and classifying *all* physical lines. (This step becomes even more critical when preparing definitions for counting logical source statements.) Include copies of pages 3 through 5 with the completed checklists you give to all who use your basic definition.
5. Complete and attach the Rules for Counting Physical Source Lines (Figure 7-1) to your definition checklist. Use a separate rules form for each source language you measure. (This step, but with the rules form for logical source statements, is even more crucial when constructing an operational definition for counting logical source statements.)
6. Use data recording forms like those in Chapter 8, either electronically or on paper, to record your measurement results. Alternatively, use your data specifications (step 3 above) as guidelines to construct specialized forms that better fit your recording needs.
7. Ask your measurement users to use definition checklists or data summary request forms to specify the reports they want to receive. Remind them to stay consistent with the definition and data recording specifications that were agreed to in steps 1 through 3.
8. Generate the reports your users request. Attach completed copies of your definition checklist, data specification checklists, and supplemental rules forms to these reports. Complete and attach also a copy of the Practices Used to Identify Inoperative Elements (Figure 7-3). This will help measurement users judge for themselves how they will interpret the results you report.

## 12.3. Report Separate Results for Questionable Elements

When applying any definition of size to a software object, perhaps the most important rule to remember is:

**When in doubt as to whether to include a particular element in a statement count, measure and report it separately.**

Then, whether you elect to include such elements in your count or not, others will know just what you have done, and they will be able to adjust your counts to meet their needs.

This advice applies especially to objects of source code that you are unsure about including in a total measure for size. Remember—you can always apply your definition for size to individual objects one at a time. If you report these results separately, your users and customers can then make their own judgments as to whether to include these particular measurement results in the totals they wish to use.

Please note that our advice does not require excluding separately measured elements from size counts. The rule is satisfied whenever you include any element in a count, so long as you make the inclusion quantitatively visible. The important point is to provide the information that others must have to intelligently interpret measurement results. Should the situation seem to require a large number of distinctly separate measures, we encourage you to think your definitions and data specifications through carefully. Proliferations of collected data sometimes add more to confusion than to value.

## 12.4. Look for Opportunities to Add Other Measures

Once you have counts for physical source lines well implemented, you may be ready to add other measures of software size. Our preferences, roughly in order, would be for counts of software units, logical source statements, and (where appropriate) function points.

Of these, we think counts of software units are likely to be the easiest to implement. Most organizations use this measure today in one form or another. We suspect that counts of software units could benefit from formal definition, perhaps much like the one in this report.

Our second choice would be to pursue counts of logical statements. Most of the work to make these counts structured and consistent has been completed in this report. The step that remains is, nevertheless, not as simple as it first seems. The missing ingredients are the detailed lists of language-specific counting rules that must be identified and recorded for each programming language. Forms like Figure 7-2 (Rules for Counting Logical Source Statements) and pages 3 through 5 of the definition checklist provide a structured framework for defining these ingredients. However, until these detailed rules have been recorded and evaluated for completeness, consistency, and acceptability to others, we are reluctant to urge adoption of logical source statement counts as a widespread standard.

In the meantime, and even before formal measurement definitions are fully implemented within your organization, applying the size checklist retroactively to describe and record the rules used to collect historical data can help you better understand the information you already have, so that you can use that information to more effectively estimate, plan, and manage current projects. This observation applies to historical counts of both physical and logical source statements. The results can be of immediate benefit, both to projects that are already underway and to those that are being planned.

## 12.5. Use the Same Definitions for Estimates that You Use for Measurement Reports

Definitions like those we have illustrated apply to estimates just as they do to actual counts. They also apply whether or not the estimates are made before or after code is produced.

Moreover, the definitions apply even when reported results are mixtures of measured and estimated values. Recognizing this is especially important when using measures of size to track design progress, and when practices and tools for obtaining exact measurements for some attribute values are less than fully developed. For example, distinguishing modified statements from programmed and copied statements are instances where estimates must sometimes be intermixed with actual counts. If counting tools and coding practices do not yet support automated identification of modified statements, best results are likely to be obtained by counting total statements (programmed plus modified, or copied plus modified) and then estimating the number of modified statements as a percentage or proportion of the total.

## 12.6. Dealing with Controversy

Source lines of code (SLOC) can be a controversial measure. Like many instruments, it stands accused of being too one-dimensional and far too easy to misuse. We agree that the world of software has many dimensions, that SLOC is but one, and that it is not the only measure we would use for software size. We also agree that we would never use SLOC as a measure without simultaneously observing and reporting other product and process characteristics. But we submit that SLOC is a useful measure, provided we understand what it represents, use it consistently, and communicate it clearly.

When we hear criticism of SLOC as a software measure, we are reminded of a perhaps apocryphal story about a ditch digger who, when asked one day how he was doing, replied, “Dug seventeen feet of ditch today.” He didn’t bother to say how wide or how deep, how rocky or impenetrable the soil, how obstructed it was with roots, or even how limited he was by the tools he was using. Yet his answer conveyed information to his questioner. It conveyed even more information, we suspect, to his boss, for it gave him a firm measure to use as a basis for estimating time and cost and time to completion.

We submit that software organizations can be at least as proficient at using this kind of information as ditch diggers—particularly if supported by the consistency and clarity made possible by well-designed definition checklists.

With respect to misuse, we can only say that if you work with customers or organizations that measure SLOC but use it in inappropriate ways, we hope that by publishing this report we can at least encourage them to misuse it consistently, for then they will have a foundation for improvement.

Finally, if your organization already measures and uses source lines of code or logical source statements intelligently, we hope we have provided you with some operational methods for improving the consistency and clarity with which you collect and report the information you use.

## 12.7. From Definition to Action—A Concluding Note

The power of clear definitions is not that they require action, but that they set goals and facilitate communication and consistent interpretation. With this report, we seek only to bring clarity to definitions. Implementation and enforcement, on the other hand, are different issues. These are action-oriented endeavors, best left to agreements and practices to be worked out within individual organizations or between developers and their customers. We hope that the materials in this report give you the foundation, framework, and operational methods to make these endeavors possible.

# References

| | |
|---|---|
| [Ada 83] | *Reference Manual for the Ada Programming Language* (ANSI/MIL-STD-1815A-1983). Washington, D.C.: United States Department of Defense, 1983. |
| [Baumert 92] | Baumert, John H. *Software Measures and the Capability Maturity Model* (CMU/SEI-92-TR-25). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992. |
| [Betz 92] | Betz, Henry P.; & O'Neill, Patrick J. *Army Software Test and Evaluation Panel (STEP) Software Metrics Initiatives Report*. Aberdeen Proving Grounds, Md.: U.S. Army Materiel Systems Analysis Activity, 1992. |
| [Boehm 81] | Boehm, Barry W. *Software Engineering Economics*. Englewood Cliffs, N.J.: Prentice-Hall, 1981. |
| [Boehm 88] | Boehm, Barry W. "A Spiral Model of Software Development and Enhancement." *Computer 8, 5* (May 1988): 61-72. |
| [Boehm 89] | Boehm, Barry W.; & Royce, Walker. "TRW IOC Ada COCOMO: Definitions and Refinements," *Proceedings of The Fifth International COCOMO Users' Group Meeting*. Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, October 1989. |
| [COBOL 85] | *American National Standard, Programming Language COBOL* (ANSI X3.23-1985 [ISO 1989-1985]). New York, N.Y.: American National Standards Institute, 1985. |
| [DOD-STD-2167A] | *Military Standard, Defense System Software Development* (DOD-STD-2167A). Washington, D.C.: United States Department of Defense, 1988. |
| [FORTRAN 77] | *American National Standard, Programming Language FORTRAN* (ANSI X3.9-1978 [ISO 1539-1980 (E)]). New York, N.Y.: American National Standards Institute, 1987. |
| [Grady 87] | Grady, Robert B.; & Caswell, Deborah L. *Software Metrics: Establishing a Company-Wide Program*. Englewood Cliffs, N.J.: Prentice-Hall, 1987. |
| [Humphrey 89] | Humphrey, Watts S. *Managing the Software Process*. Reading, Mass.: Addison-Wesley, 1989. |
| [Ichbiah 86] | Ichbiah, Jean D.; Barnes, John G. P.; Firth, Robert J.; & Woodger, Mike. *Rationale for the Design of the Ada Programming Language*. Minneapolis, Minn.: Honeywell Systems and Research Center, 1986. |
| [IEEE 90] | *IEEE Standard Glossary of Software Engineering Terminology* (IEEE Std 610.12-1990). New York, N.Y.: The Institute of Electrical and Electronics Engineers, Inc., 1990. |

|  |  |
|---|---|
| [IEEE 92] | *Standard for Software Productivity Metrics [draft]* (P1045/D5.0). Washington, D.C.: The Institute of Electrical and Electronics Engineers, Inc., 1992. |
| [IFPUG 91] | International Function Point Users Group. *Function Point Counting Practices Manual, Release 3.2.* Westerville, Ohio: IFPUG, 1991. |
| [Kernighan 88] | Kernighan, Brian W.; & Ritchie, Dennis M. *The C Programming Language, Second Edition*. Englewood Cliffs, N.J.: Prentice-Hall, 1988. |
| [McGarry 90] | McGarry, John J. “Applied Software Metrics” (Presentation Charts). Newport, R.I.: Naval Underwater Systems Center, 1990. |
| [McGhan 91] | McGhan, James N.; & Dyson, Peter B. *CECOM Executive Management Software Metrics (CEMSM)—CEMSM Guidebook*. Indialantic, Fla.: Software Productivity Solutions, Inc. 1991. |
| [McGhan 92] | McGhan, James N.; & Dyson, Peter B. *CECOM Executive Management Software Metrics (CEMSM)—Cost Benefit Analysis*. Indialantic, Fla.: Software Productivity Solutions, Inc. 1992. |
| [Park 88] | Park, Robert E. “The Central Equations of PRICE S,” *Proceedings of the Fourth Annual COCOMO Users’ Group Meeting*. Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, November 1988. |
| [Putnam 91] | Putnam, Larry H. *Measures for Excellence: Reliable Software On Time, Within Budget*. Englewood Cliffs, N.J.: Prentice-Hall, 1991. |
| [Rozum 92] | Rozum, James A. *Software Measurement Concepts for Acquisition Program Managers* (CMU/SEI-92-TR-11). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1992. |
| [Schultz 88] | Schultz, Herman P. *Software Management Metrics* (ESD-TR-88-001). Bedford, Mass.: The MITRE Corporation, 1988. |
| [USAF 92] | *Software Management Indicators* (Air Force Pamphlet 800-48). Washington, D.C.: Department of the Air Force, 1992. |
| [Stroustrup 87] | Stroustrup, Bjarne; *The C++ Programming Language*, Reading, Mass.: Addison-Wesley, 1987. |

# Appendix A: Acronyms and Terms

## A.1. Acronyms

| | |
|---|---|
| **CMU** | Carnegie Mellon University |
| **COCOMO** | Constructive Cost Model [Boehm 81] |
| **COTS** | commercial off-the-shelf |
| **CSC** | computer software component |
| **CSCI** | computer software configuration item |
| **CSU** | computer software unit |
| **DARPA** | Defense Advanced Research Projects Agency |
| **DoD** | Department of Defense |
| **DSI** | delivered source instructions |
| **DSLOC** | delivered source lines of code |
| **GFS** | government furnished software |
| **IEEE** | The Institute of Electrical and Electronics Engineers, Inc. |
| **IFPUG** | The International Function Point Users Group |
| **KDSI** | thousands of delivered source instructions |
| **KLOC** | thousands of lines of code |
| **KSLOC** | thousands of source lines of code |
| **LOC** | lines of code |
| **PDL** | program design language |
| **RFP** | request for proposal |
| **SEI** | Software Engineering Institute |
| **SLOC** | source lines of code |
| **SWAP** | Software Action Plan |
| **3GL** | third-generation language |
| **4GL** | fourth-generation language |

## A.2. Terms Used

**Attribute** - A quality or characteristic of a person or thing. Attributes describe the nature of objects measured.

**Blank comments** - Source lines or source statements that are designated as comments but contain no other visible textual symbols.

**Blank lines** - Lines in a source listing or display that have no visible textual symbols.

**Comments** - Textual strings, lines, or statements that have no effect on compiler or program operations. Usually designated or delimited by special symbols. Omitting or changing comments has no effect on program logic or data structures.

**Compiler directives** - Instructions to compilers, preprocessors, or translators. Usually designated by special symbols or keywords.

**Computer software component (CSC)** - A distinct part of a computer software configuration item (CSCI). CSCs may be further decomposed into other CSCs and computer software units (CSUs) [DOD-STD-2167A].

**Computer software configuration item (CSCI)** - A configuration item for software [DOD-STD-2167A].

**Computer software unit (CSU)** - An element specified in the design of a computer software component (CSC) that is separately testable [DOD-STD-2167A].

**Continue statement** - Statements that have no effect on a program's logic other than to pass to the next statement. In some languages (FORTRAN is an example), labels can be attached to *continue* statements so that they can be used as destinations for or terminations of logical execution paths.

**Dead code** - Code that is present in a delivered product but is never referenced, accessed, or used.

**Declarations** - A non-executable program statement that affects the assembler's or compiler's interpretation of other statements in the program. Examples include type and bounds declarations, variable definitions, declarations of constants, static initializations, procedure headers and argument lists, function declarations, task declarations, package declarations, interface specifications, generic declarations, and generic instantiations.

**Delivered statements** - Statements that are delivered to a customer as part of or along with a software product. There are two subclasses: (1) statements delivered in source form and (2) statements delivered in executable form but not as source.

**Embedded statement** - A statement used within or as an argument to another or inserted between begin/end markers.

**Empty statement** - A statement that occurs when two or more statement delimiters appear in succession. This definition holds even when the delimiters are separated by one or more blank characters or when the delimiters are on different source lines.

**Executable statement** - A statement that produces runtime actions or controls program flow.

**Format statement** - A statement that provides information (data) for formatting or editing inputs or outputs.

**Logical source statement** - A single software instruction, having a defined beginning and ending independent of any relationship to the physical lines on which it is recorded or printed. Logical source statements are used to measure software size in ways that are independent of the physical formats in which the instructions appear.

**Master source statements** - The source statements to which changes or additions are made when maintenance actions are needed.

**Measure** - *n*. A standard or unit of measurement; the extent, dimensions, capacity, etc. of anything, especially as determined by a standard; an act or process of measuring; a result of measurement. *v*. To ascertain the quantity, mass, extent, or degree of something in terms of a standard unit or fixed amount, usually by means of an instrument or process; to compute the size of something from dimensional measurements; to estimate the extent, strength, worth, or character of something; to take measurements.

**Measurement** - The act or process of measuring something. Also a result, such as a figure expressing the extent or value that is obtained by measuring.

**Nondelivered statements** - Statements developed in support of the final product, but not delivered to the customer.

**No-op statement** - Statements that have no effect on a program's logic other than to pass to the next statement. In assembly languages, no-op statements—unlike null statements—can consume execution time.

**Null statement** - Statements that have no effect on a program's logic other than to pass to the next statement.

**Origin** - An attribute that identifies the prior form, if any, upon which product software is based.

**Physical replicates** - Copies of blocks or sections of master source code that are included in a product through physical storage in master source files.

**Physical source statement** - A single line of source code.

**Postproduction replicates** - Copies of a product that are generated from the same master source code and that form part of the extended operational system.

**Source statements** - Instructions or other textual symbols, either written by people or intended to be read by people, that direct the compilation or operation of a computer-based system.

**Statement delimiter** - A symbol or set of rules that identifies when a statement begins or ends.

**Statement type** - An attribute that classifies source statements and source lines of code by the principal functions that they perform.

**Usage** - An attribute that distinguishes between software that is developed or delivered as an integral part of the primary product and software that is not.

# Appendix B: Candidate Size Measures

When the Size Subgroup of the Software Metrics Definition Working Group began its work, we started by constructing lists of observable inputs and outputs that might be used to quantify the size of activities associated with producing and supporting software systems. We found far more potential targets for measurement than we had thought possible. Figure B-1 shows our results. We present the lists here not just because they served as our starting point, but so that you can use them to help launch explorations of other measures of size that can be used to manage and improve software processes. We make no claim that the lists are complete—they merely note some of measures worth considering.

In constructing the lists in Figure B-1, we viewed activities as processes that take inputs and transform them into outputs. The thesis was that by measuring the volume of inputs and outputs, organizations gain information that helps them plan, manage, and improve their underlying processes.

Although our focus is on processes, the measurable elements in Figure B-1 are almost always *work products*, not activities. Whenever our goals are to plan, control, and improve processes, one of our principal methods for attaining these goals is by monitoring products. As size measurement evolves, it is from lists like those in the figure that new candidates for size measures can be selected. We believe that checklist-based methods like the ones in this report can be effective in defining many of these potential measures.

One reviewer expressed concern that Figure B-1 could suggest a fixation on a waterfall model of software development. This was certainly not the case when the Size Subgroup generated the lists. We simply identified activities frequently used in producing and supporting software systems, regardless of the process model used. The names in the left column are representative of those used in references like DOD-STD-2167A and IEEE Std 610.12-1990. Readers who find other terms better suited for describing their own local practices should feel free to modify the activity names in Figure B-1.

| Activity | Measurable Inputs | Measurable Outputs |
|---|---|---|
| **Requirements Analysis** | verbs | pages of design requirements |
| | shalls | objects |
| | requirements units | object classes |
| | RFP pages | bubbles |
| | proposal pages | CSCIs |
| | characters of text | interface control documents |
| | | rules |
| | | Petri net firings |
| | | threads (of control) |
| | | external interfaces |
| | | function points |
| | | function point primitives: files, records, fields, transactions, inquiries, etc. |
| | | entries into the requirements traceability matrix |
| | | parts of requirements: screens, displays, files, panels, messages, etc. |
| | | system report formats |
| | | input screen formats |
| | | peer reviews |
| | | inspections |
| **Test Planning** | shalls | test steps |
| | requirements units | test cases |
| | CSCIs | unique tests |
| | rules | peer reviews |
| | threads (of control) | inspections |
| | external interfaces | |
| | mappings from requirements | |
| | inter-system interfaces | |

Figure B-1 Software Size—A Partial Listing of Targets for Measurement

| Activity | Measurable Inputs | Measurable Outputs |
|---|---|---|
| **Design** | pages of design requirements<br>characters of text in requirements<br>objects<br>classes<br>instance relationships<br>rendezvous<br>bubbles<br>CSCIs<br>rules<br>Petri net firings<br>external interfaces<br>external (user) inputs<br>function points<br>system report formats<br>input screen formats | CSCs and CSUs<br>lines of PDL<br>pages of coding specifications<br>pages of documentation<br>objects<br>bubbles<br>charts<br>frames<br>asynchronous controls<br>synchronous controls<br>tasks<br>rendezvous<br>threads<br>message relationships<br>stimulus-response relationships<br>internal interfaces<br>intrasystem interfaces<br>mappings from requirements<br>algorithms<br>encapsulated operations<br>design walkthroughs<br>design inspections<br>prototype screens, displays, files, panels, messages, etc. |
| **Coding & Unit Testing** | objects<br>bubbles<br>charts<br>frames<br>asynchronous controls<br>synchronous controls<br>CSCs & CSUs<br>tasks<br>rendezvous<br>internal interfaces<br>mappings from design<br>algorithms<br>encapsulated operations<br>builds<br>deliveries<br>increments | source statements<br>source lines<br>source characters<br>noncommentary source words<br>COBOL verbs<br>pages of code<br>pages of documentation<br>comments<br>objects<br>CSUs<br>tasks<br>machine language instructions<br>code walkthroughs<br>code inspections<br>code paths<br>exception conditions<br>bytes of compiled code<br>bits of compiled code<br>words of memory |

Figure B-1 Software Size—A Partial Listing of Targets for Measurement (Continued)

| Activity | Measurable Inputs | Measurable Outputs |
|---|---|---|
| **Integration & Testing** | source statements<br>source lines<br>source characters<br>noncommentary source words<br>COBOL verbs<br>CSCs & CSUs<br>test steps<br>test cases<br>unique tests<br>inter-module interfaces<br>inter-system interfaces<br>external interfaces<br>exception conditions<br>branches<br>cause-effect pairs<br>deliveries | source statements<br>source lines<br>source characters<br>noncommentary source words<br>bytes in load modules<br>COBOL verbs<br>CSCs & CSUs<br>test steps<br>test cases<br>unique tests<br>inter-module interfaces<br>inter-system interfaces<br>external interfaces<br>test results<br>faults<br>function points<br>test coverage<br>thread tests<br>peer reviews/inspections |
| **Operations** | source statements<br>source lines<br>source characters<br>noncommentary source words<br>COBOL verbs<br>operators<br>operands<br>nesting levels<br>pages of code<br>pages of user documentation<br>pages of maintainer documentation<br>test steps<br>test cases<br>unique tests<br>inter-system interfaces<br>exception conditions<br>job steps<br>problem reports<br>change requests<br>function points | run times<br>memory requirements<br>number of nodes<br>number of terminals<br>number of replicated systems<br>number of statements bypassed<br>people served<br>sites served<br>number of installations<br>average daily throughput volume<br>number of reports generated per week/month<br>user/operator actions<br>user/operator keystrokes<br>problems closed<br>change requests completed |

Figure B-1 Software Size—A Partial Listing of Targets for Measurement (Continued)

| Activity | Measurable Inputs | Measurable Outputs |
|---|---|---|
| **Peer Reviews** | source statements<br>source lines<br>noncommentary source words<br>proposal pages<br>plan pages<br>report pages<br>people assigned<br>test steps<br>test cases | defects found<br>improvement opportunities<br>action items |
| **Documentation** | outlines<br>drafts<br>  pages of text<br>  tables<br>  figures<br>names and organizations on<br>  distribution lists | pages<br>final documents<br>  electronic<br>  printed<br>  collated<br>  assembled<br>  distributed<br>copies<br>reviews<br>change bars<br>peer reviews<br>inspections |
| **Quality Assurance** | standards<br>policies<br>procedures<br>processes | audits<br>reports<br>action items<br>tests witnessed<br>tests performed<br>formal reviews<br>informal reviews<br>walkthroughs<br>peer reviews<br>meetings<br>products & summaries from reviews,<br>  walkthroughs, etc. |
| **Configuration Management** | software products<br>problem reports<br>change proposals<br>people on the change control board | reports<br>baselines<br>change notices<br>approvals<br>meetings<br>items resolved |
| **Change Control Board Meetings** | problem reports<br>change proposals | change notices<br>approvals<br>disapprovals<br>deferrals |

Figure B-1 Software Size—A Partial Listing of Targets for Measurement (Continued)

Readers who would like to expand the lists in Figure B-1 (or create others of a similar nature) should keep in mind that outputs of one process or activity are almost always inputs to others. An important corollary is that inputs come from somewhere, and that "somewhere" is often another activity. Figure B-2 is a simple illustration. Explicit recognition of this principle helped the Size Subgroup significantly, not only when building the lists of inputs and outputs, but also when identifying the processes (activities) that are producers and consumers of software-related artifacts. By consciously tracing the flows of inputs and outputs, members found many other inputs, outputs, and activities that they might otherwise have overlooked.

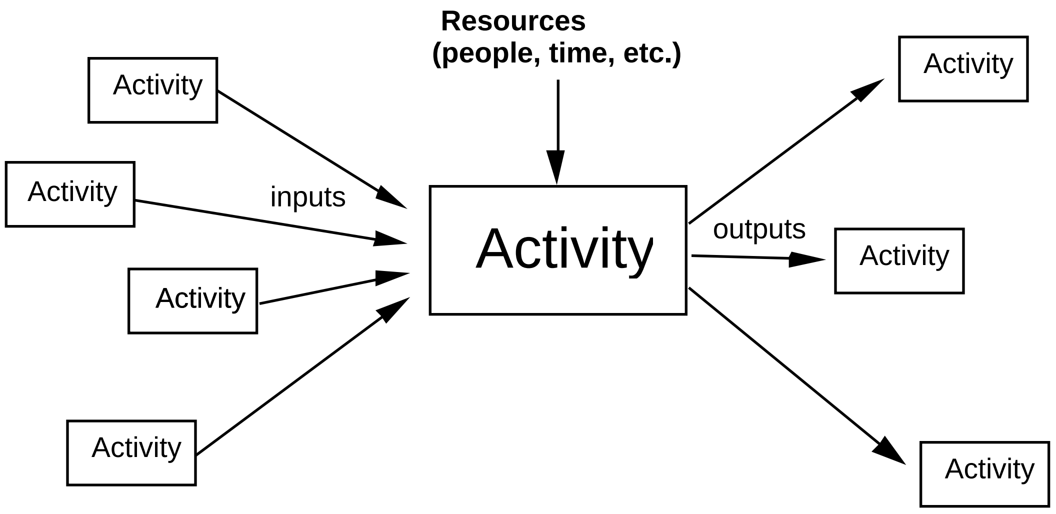


Figure B-2 Using Inputs and Outputs to Identify Activities

Another point is also worth noting. Sometimes the lists in Figure B-1 appear to show the same items as both inputs and outputs to a single activity. On reflection, it should become apparent that although the names are the same, the items themselves are different. For example, bubbles and objects are listed as both inputs and outputs of software design. The difference is that output bubbles and objects are expanded versions that contain more detail than those that were received as input specifications. Although of the same class, they are not the same items, as they represent different levels of design decomposition. Similarly, the inputs and outputs for integration and test share several common names (for example, source statements, source lines, CSCs and CSUs, test steps, and inter-module interfaces). The difference here is that the outputs are integrated and tested items, while the inputs are not. Why might we be interested in the distinctions between such measures? One reason is that differences between observed volumes of inputs and outputs can be useful for measuring backlogs and for tracking progress, particularly when compared against corresponding values in project plans.

# Appendix C: Selecting Measures for Definition—Narrowing the Field

The checklist-based framework in this report appears to be applicable to many potential size measures. Nevertheless, each measure requires special treatment. To progress from candidate measures to specific examples, the Size Subgroup had to sharpen its focus. In this appendix we explain our reasons for selecting source statement counts as the basis for illustrating the framework, and we outline the path we followed in specializing the framework to these measures.

## C.1. The Choices Made

The Size Subgroup chose two source code measures—physical source lines and logical source statements—as our first measures for formal definition. The criteria that most influenced these selections were:

– **Frequency of use (and misuse).** The primary goal from the beginning of this effort was to clarify measures that are used widely now, but are used with inconsistency and ambiguity.

– **Utility with respect to the Capability Maturity Model [Humphrey 89].** We concluded that we should focus on fundamental management measures that can help organizations move from level 1 (ad hoc) to level 2 (managed) processes, yet still remain useful as process maturities increase.

– **Timeliness.** There were strong advocates in the Subgroup for pushing size measurement back as early into the development process as possible.

– **Usefulness as predictors.** We recognized that the value of size measures is governed by our abilities to use them as predictors when planning and managing downstream activities.

– **Automatability.** We believed that the ability to automate size measurement will be essential for satisfying other criteria such as economy, consistency, and completeness, as well as for overcoming resistance to change.

In retrospect, the fourth criterion was probably the most influential in helping us assign priorities to our most highly rated measures. It says that in order to gain confidence in a size measure, we must be able to evaluate its effectiveness in terms of other, downstream measures. This in turn implies that downstream measures must already be in place. Since the subgroup could not hope to define all size measures on the first pass, our conclusion was that we should begin with measures whose utility as predictors of cost and schedule are already established. By doing so, we could lay a foundation from which to begin working back upstream.

Figure C-1 is a sketch of one sequence we discussed. Although the flow of quantitative information begins in the upper left corner, it would be a mistake to start there when constructing our first definitions. The reason is that effectiveness of size measures is usually best judged in terms of the help they provide in predicting the sizes of subsequent stages. Without downstream definitions, we have no criteria for testing the predictive capabilities of upstream measures. By starting at the lower right and working upstream, we ensure that tools (defined measures) are available to evaluate upstream measures, so that we can turn measurement results into estimates and plans of verifiable quality.

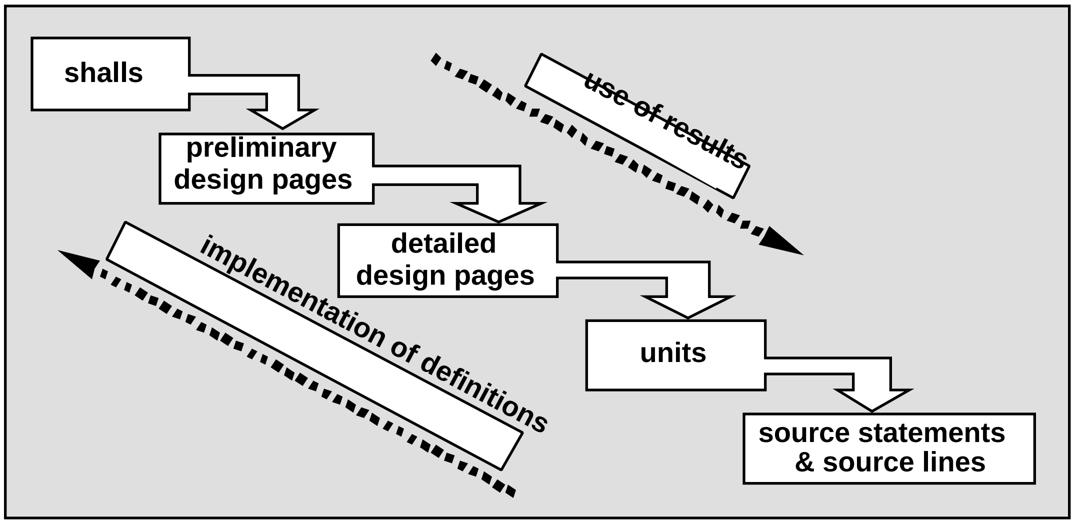


Figure C-1 Priorities for Definition

Source lines and statements are not the only size measures used today for planning and estimating software activities. In some environments, especially those in which business applications are developed or used, function points are often preferred over source lines and source statements to describe product size. Function points also ranked high on the Size Subgroup's list for early definition. They were deferred not because they were viewed as unimportant, but because they did not satisfy our criterion for automatability, because they are not particularly suitable for project tracking, and because another organization, the International Function Point Users' Group (IFPUG), has assumed jurisdiction. Since IFPUG is working actively to formalize practices for function point measures [IFPUG 91], the Size Subgroup saw little value in undertaking duplicative efforts.

## C.2. The Path Followed

Figure C-2 outlines the path initiated by the Size Subgroup and followed since by the SEI in preparing this report. As indicated, the first class of measures to which we applied the framework was counts of source code statements. Within this class, two particular measures have been addressed—physical source lines and logical source statements. The difference between the two is that lines of code are physical entities, while logical statements are attempts to characterize software size in ways that are independent of the physical formats in which instructions appear.

We constructed a definition checklist for these two size measures. We then used the checklist to create two example definitions, one for physical source lines and the other for logical source statements. We also used the checklist to illustrate some recording and reporting specifications that might be appropriate for organizations with different measurement needs and capabilities.

Although the examples created to date center around source code measures (counts of source lines and source instructions), our view is that these are but two of many useful measures of software size. Figure B-1 lists many other possibilities. The Size Subgroup selected source lines and source instructions for its first examples primarily because counts of these entities are among the most widely used (and misused) software size measures today. Source lines and, to a lesser extent, source instructions are also measures for which automated counters can readily be constructed, and they are clearly measures that can benefit from rules that explain exactly what is included in (and what is excluded from) reported results.

By providing example definitions for counting physical and logical source statements, we are in no way recommending them as the only measures of size. Given time, we would define other measures as well.

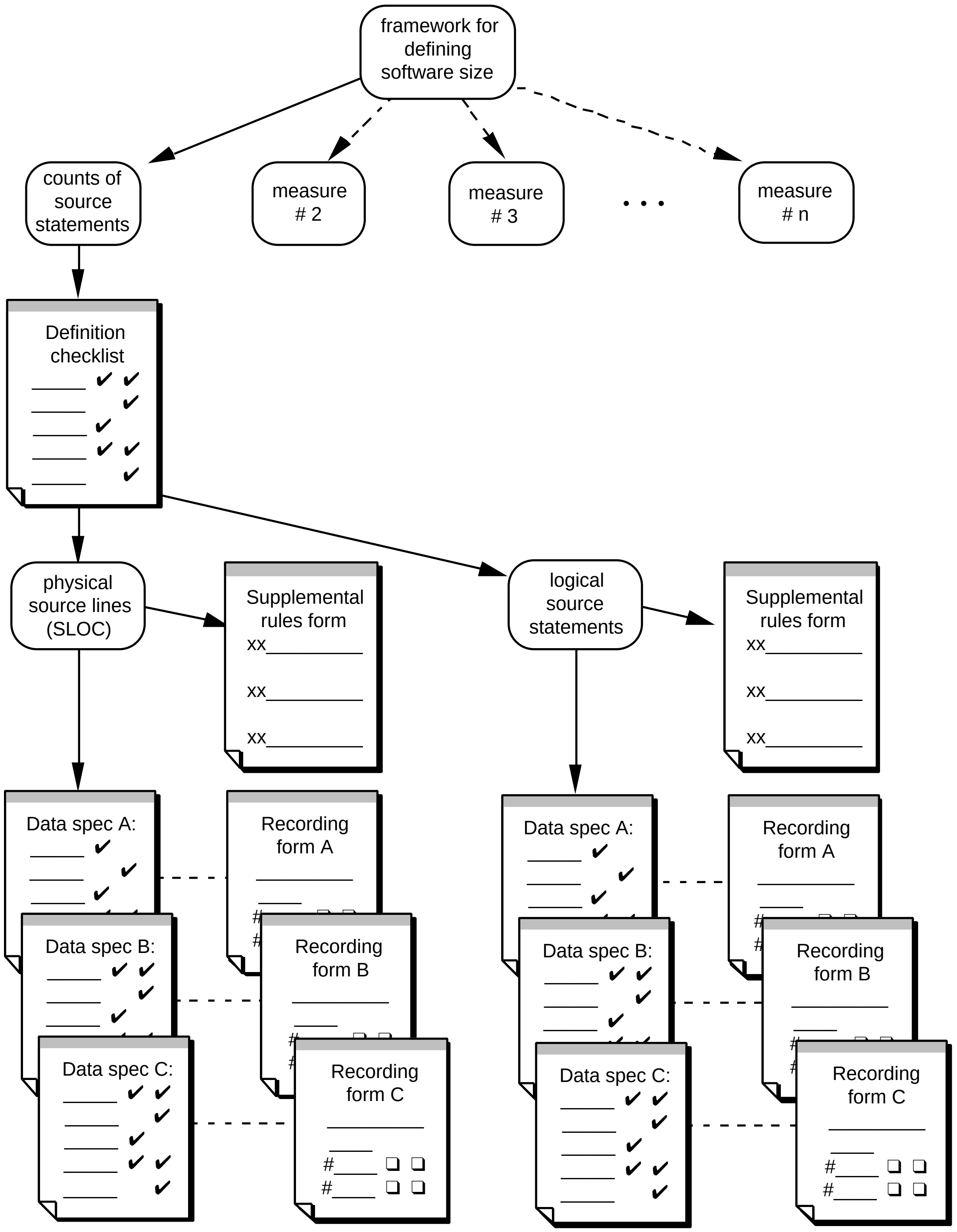


Figure C-2 The Path Followed When Applying the Framework to Construct Specific Definitions and Data Specifications

# Appendix D: Using Size Measures—Illustrations and Examples

In this appendix, we illustrate a few of the ways in which we have seen counts of source statements used to help plan, manage, and improve software projects and processes. Our purpose is not to be exhaustive, but rather to highlight some interesting uses that you may find worth trying in your own organization. We also want to encourage organizations to seek other new and productive ways to put size measures to work, and we would very much like to hear from those who succeed. We would like to see other examples of interesting and creative applications of size measurement, and we invite you to send your success stories to us.

Like most good ideas, the ones we show here have all been borrowed from someone else. In fact, that was our principal criterion: the examples in this appendix—or ideas much like them—have all been used in practice. We are particularly indebted to John McGarry of the Naval Underwater Systems Center for sharing his methods with us. Figures D-2 through D-9 are based on charts from presentations we have heard him make [McGarry 90]. Similarly, Figures D-10 and D-11 are adaptations of illustrations we found in Bob Grady's and Deborah Caswell's excellent book on experiences with software metrics at Hewlett-Packard [Grady 87].

## D.1. Project Tracking

The first use of counts of source statements that we illustrate (Figure D-1) will be familiar to almost every software professional. It is simply a display of the cumulative, month-to-month history of a project's progress through its various overlapping phases. Progress can be measured in terms of either logical or physical source statements. Charts like this are designed to be updated each reporting period, so that the points for the most recent entries reflect current status.

In the early reporting periods of a project, not all points on charts like Figure D-1 will represent actual measurements. Values for the number of statements designed but not yet coded will always be estimates, so totals for statements designed will be mixtures of estimates and actual counts. This state of affairs will persist until all coding is complete.

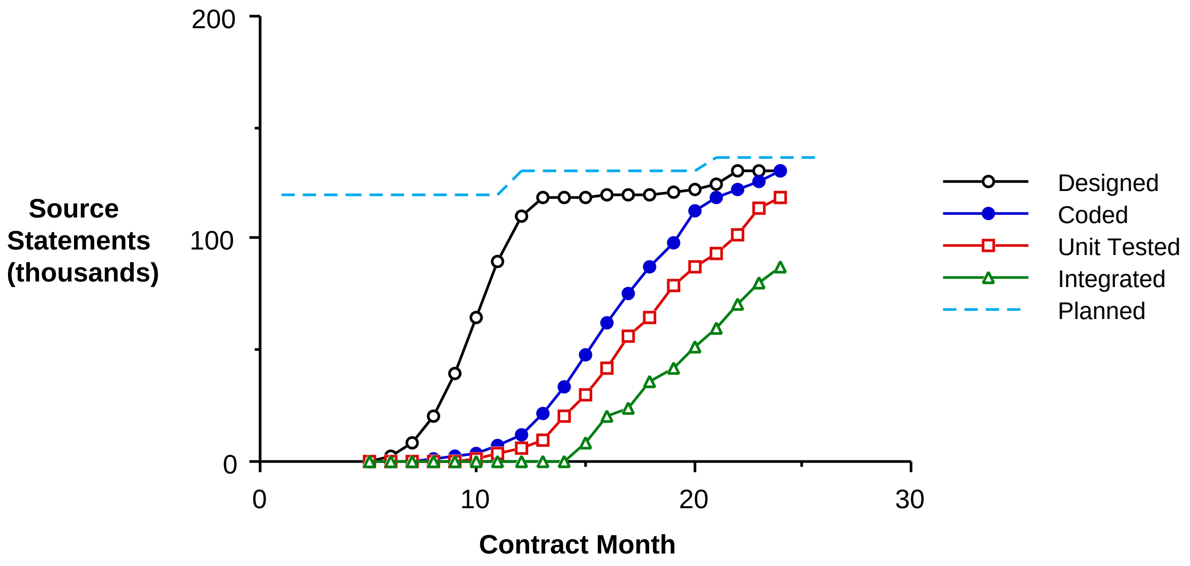


Figure D-1 Tracking Development Progress

## D.2. Exposing Potential Cost Growth

Our second illustration is interesting in that it shows an instance where definitions for size and projections of measurement results can be used long before actual measurement results become available. In Figure D-2 we show an example in which an acquisition agency (or perhaps the upper level management of a development contractor) has plotted two pieces of information extracted from each of a succession of development plans. This contract was bid and won on the basis that there would be substantial reuse of existing code. The first conclusion that one could draw from Figure D-2 is that because code growth appears to be nearing 20%, costs are likely to be similarly affected. The second conclusion is that the situation is in fact much worse—all of the promised reuse has disappeared and the forecast for new code development is now up by 50%. If this information has not been reflected in current cost estimates and schedules, some serious questions should be asked.

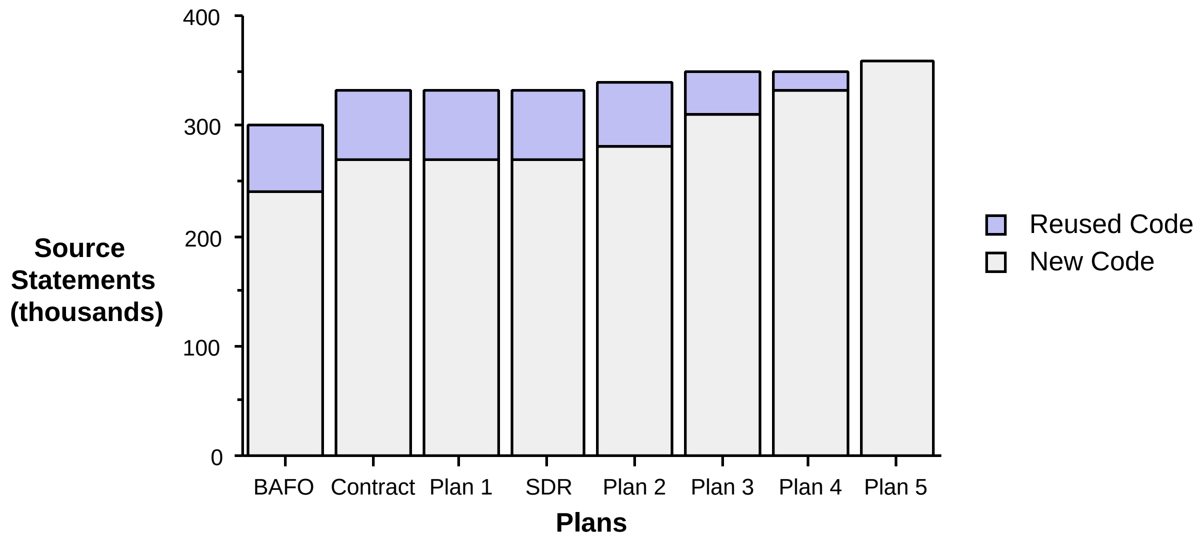


Figure D-2 Exposing Potential Cost Growth—The Disappearance of Reused Code

## D.3. Gaining Insight into Design and Coding Practices

In our next examples, Figures D-3 and D-4 illustrate the use of source statement measures to gain insight into design and coding practices. This insight can be particularly helpful when new tools and methods such as Ada or object-oriented design are introduced or when organizations want quantitative indications of the progress and effectiveness of process improvements and training.

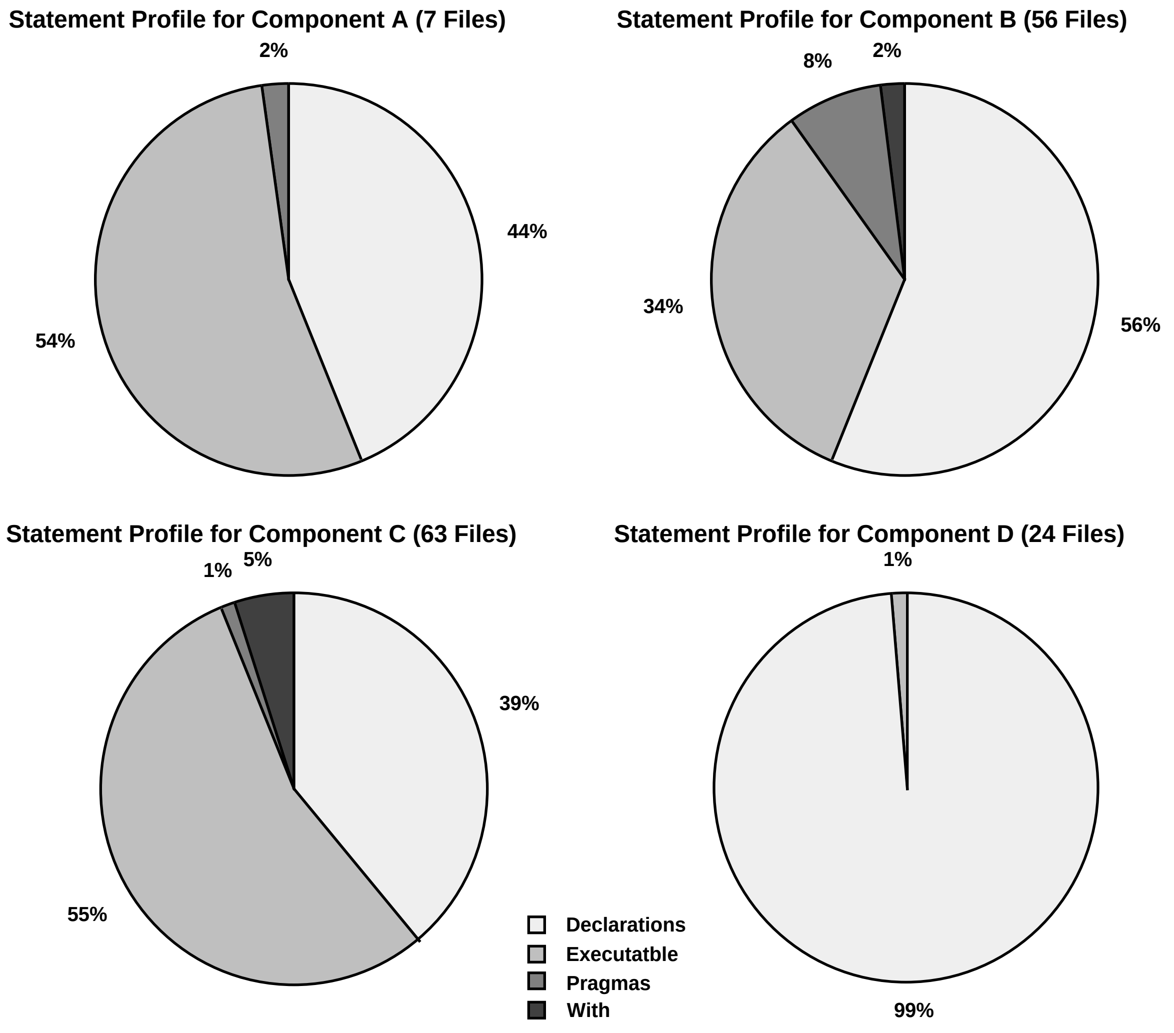


Figure D-3 Comparison of Product Components—Uncovering Poor Design Practices

For example, with Figure D-3 and the additional information that components A, B, C, and D form a product being developed in Ada, we can infer that some of the major features of Ada have not been exploited. In particular, the data for component D shows that 99% of D is composed of data declarations, suggesting that much of the product's design may be based on shared global declarations that permeate and ripple through other modules. This merits investigation, as the use of global declarations and global data is one of several programming practices that software professionals now recognize as being error-prone. Ada and other new languages have been specifically designed so that practices like this can be eliminated.

Figure D-4 provides another example that leads to much the same conclusion. Here we see a case where 40 of 47 modules in a product contain no type definitions at all, while one of the other modules has between 16 and 31. This suggests that the developers of this software have not yet learned to use the principles of information hiding that are now recognized as so important to preventing defects and to reducing development and maintenance costs. Had they used information hiding, we could expect to see a greater number of local definitions isolated within individual modules, where their effects could not inadvertently contaminate other modules.

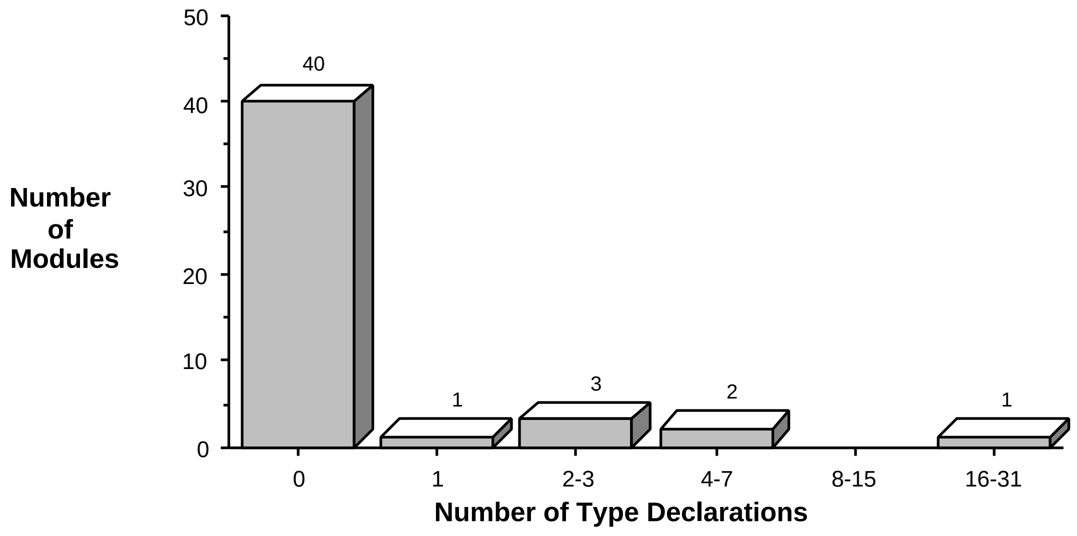


Figure D-4 Detecting Weak Designs—The Absence of Information Hiding

## D.4. Early Warnings: The Need for Schedule Replanning

Figures D-5, D-6, and D-7 show three views of the same project. Figure D-5, which shows coding progress plotted against the third version of the development plan, is of a type often used by development and acquisition managers. If this is all the information we are shown, we might infer that the project has been pretty much on schedule through month 10, but that it has now started to fall behind and may require some minor adjustments or replanning to bring it back onto track.

Figure D-6, however, suggests that the problems may be more serious than Figure D-5 indicates. Here we have plotted measured progress against the projections in the original plan, and major shortfalls are apparent. After seeing Figure D-6, we would certainly want to ask about the replanning that has occurred since the original plan and the impact that the departures from the original plan will have on projected costs and schedules.

Interestingly, we do not have to wait for actual measurements from the development organization to gain much of the insight we seek. In fact, we could have obtained this insight even earlier. As Figure D-7 shows, if we simply plot the data from each of the developer's plans on the same graph, we see that early dates have simply been slipped and that no real schedule replanning has been done.

Figure D-7 suggests some even more probing questions. Since the developer has made but minor changes in the planned completion date despite falling significantly below the original profile over the last nine months, there is reason to examine the production rate that the current plan implies. When we do this, we see that the current plan projects that code will be completed at an average rate of 12,000 statements per month for months 12 through 20. This is highly suspect, since the developer's demonstrated production capability has yet to reach an average rate of even 2,500 statements per month, something considerably less

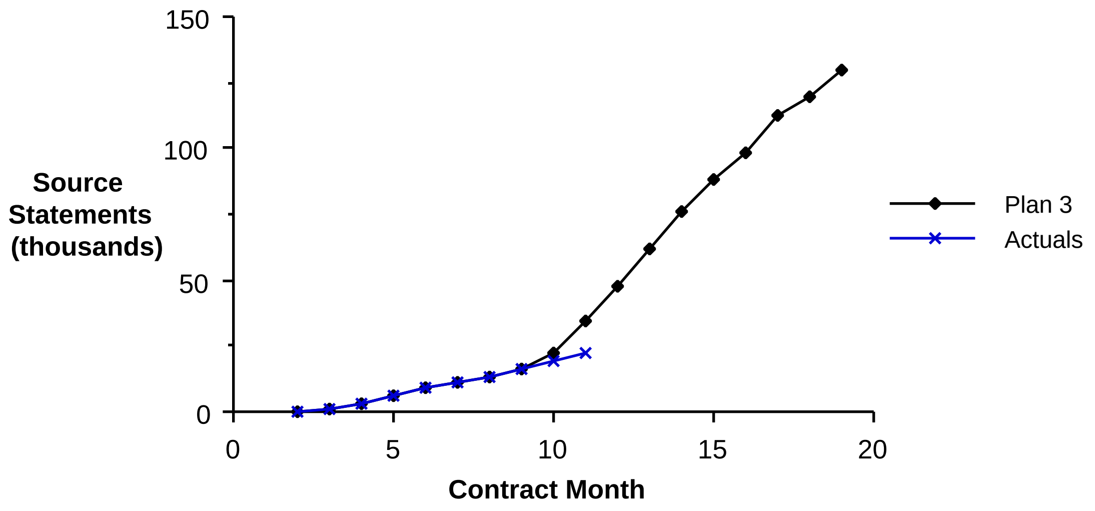


Figure D-5 Project Tracking—The Deviations May Seem Manageable

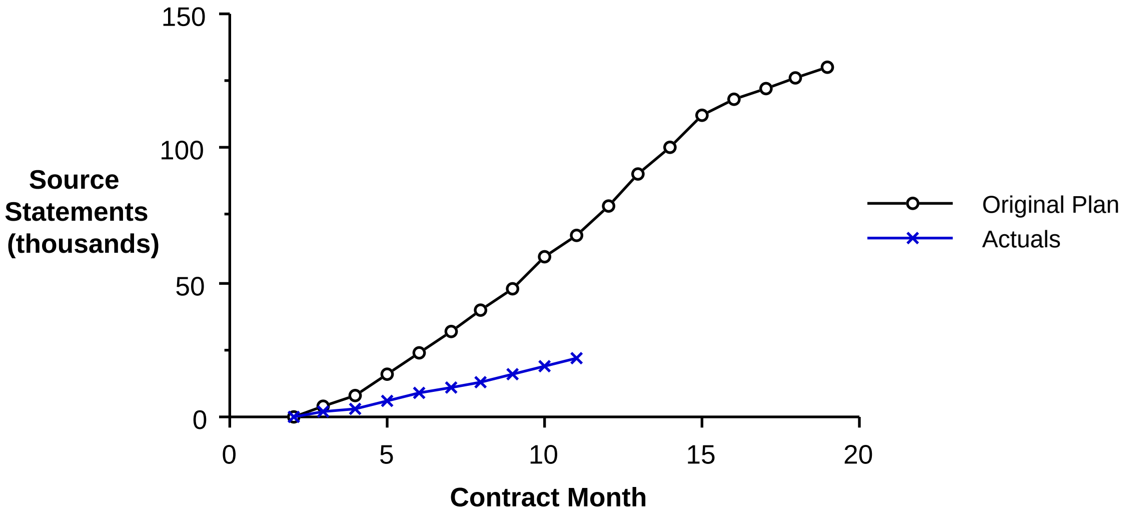


Figure D-6 Project Tracking—Deviations from Original Plan Indicate Serious Problems

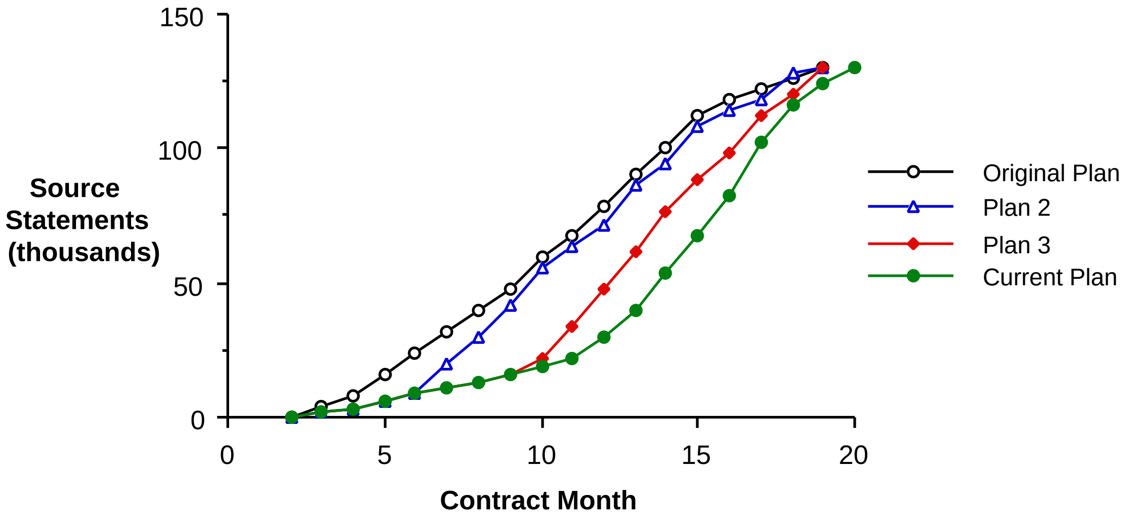


Figure D-7 Project Tracking—Comparisons of Developer’s Plans Can Give Early Warnings of Problems

than the rate of 7,600 statements per month in the original plan. It would be interesting at this point to examine the developer’s current plan to see how it proposes to more than quadruple the rate of code production. If, in fact, the developer sticks with a proposal to use accelerations of this magnitude to meet the original completion date, it may be wise to place increased emphasis on measuring and tracking the quality of the evolving product.

## D.5. Detecting Deferred Development

This example looks at changes that have occurred in plans for the first five builds of a product that is being developed in sequential stages. The first two builds depicted in Figure D-8 show no major anomalies other than the disappearance of 25,000 statements that were being counted on for reuse. Since these statements do not reappear in later builds, the presumption is that they have been replaced by new code, with concomitant implications for cost and schedule.

Of more interest, however, is that nearly 90,000 statements have been omitted or deferred from Builds 3 and 4, perhaps to reappear in Build 5, which has grown by 140,000 statements. Here we have indications not only that the size of the product is increasing, but also that the scheduled completion date is likely to be in even greater danger. We would be inclined to give the plans and estimates for Build 5 very close scrutiny.

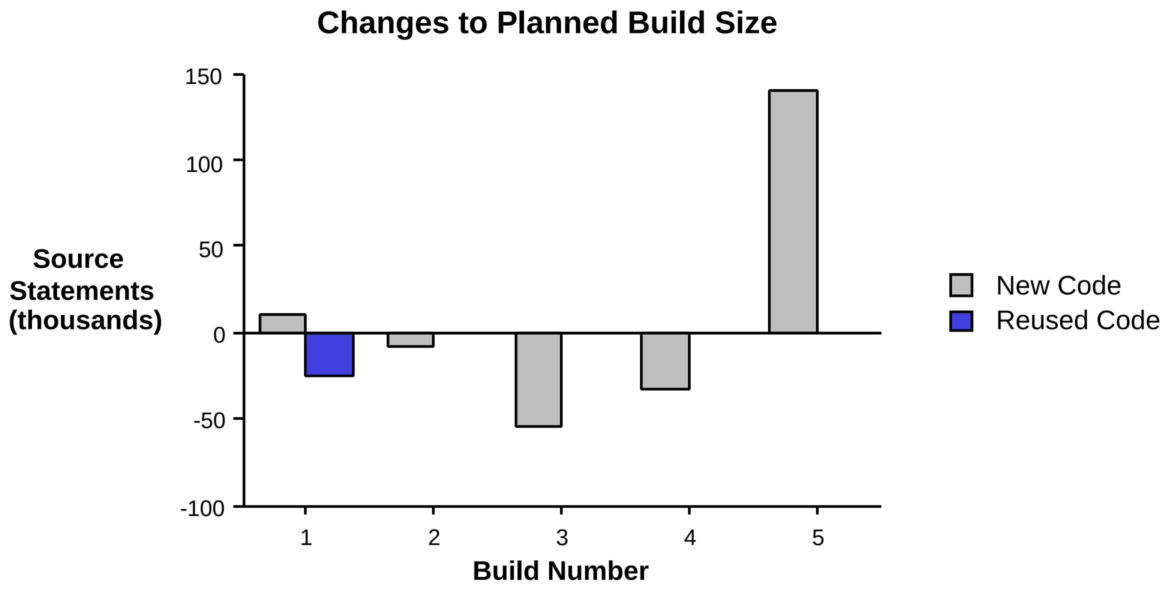


Figure D-8 Indications of Deferred Development and Disappearing Reuse

## D.6. Indications of Design Turbulence

Figure D-9 is a simple plot of the estimated sizes of the different configuration items (CSCIs) that make up a new software system. Two values are plotted for each CSCI—the estimated size at the software specification review (SSR) and the estimated size at the preliminary design review (PDR). Although the plotted values suggest some code growth (something easily confirmed with another type of chart), our major concern here is the indication of large differences between the SSR estimates and the PDR estimates for most of the configuration items. For eight of the items, the discrepancy is more than 100% of the smaller value. Apparently there has been a substantial change in either the developing organization's view of the product or its understanding of the job. Alerted by this information, although it is far from conclusive, we would want to probe further to ensure that we understand the reasons for the extent of the changes and the possible implications this has for product quality and the needs for project replanning.

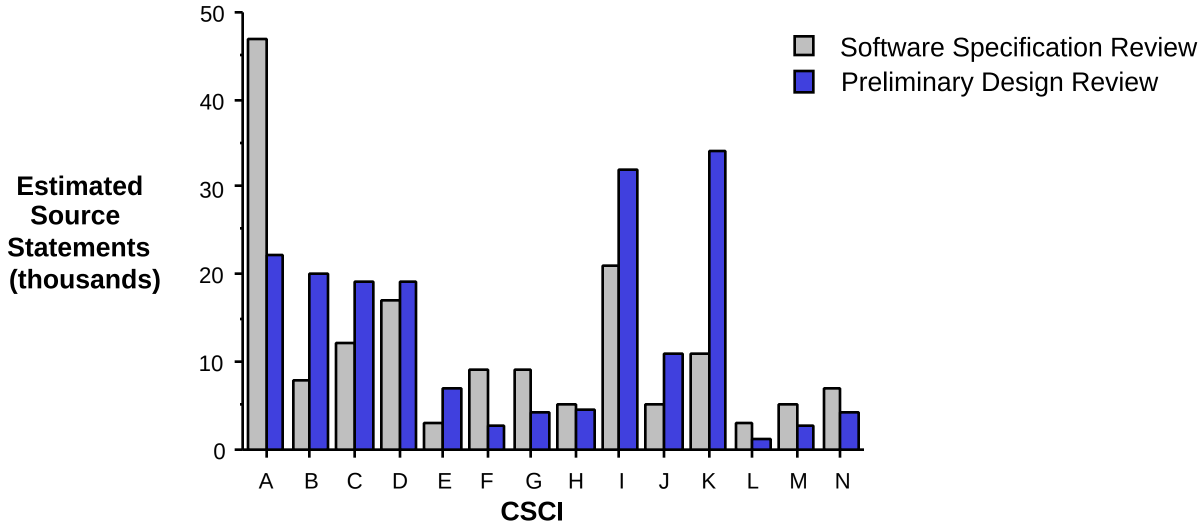


Figure D-9 Indications of Design Turbulence

## D.7. Normalizing Quality Measures

Figures D-10 and D-11 illustrate two applications of quality measures in which normalizations with respect to size play an important role. Without normalization based on appropriately defined size measures, many inferences drawn from quality measurements would be invalid.

In Figure D-10, we see how one organization's experience with discovery of software errors varies as a function of product type. This figure also shows rather dramatically one reason why much emphasis in software engineering today is directed toward increasing the amount of reuse in software development.

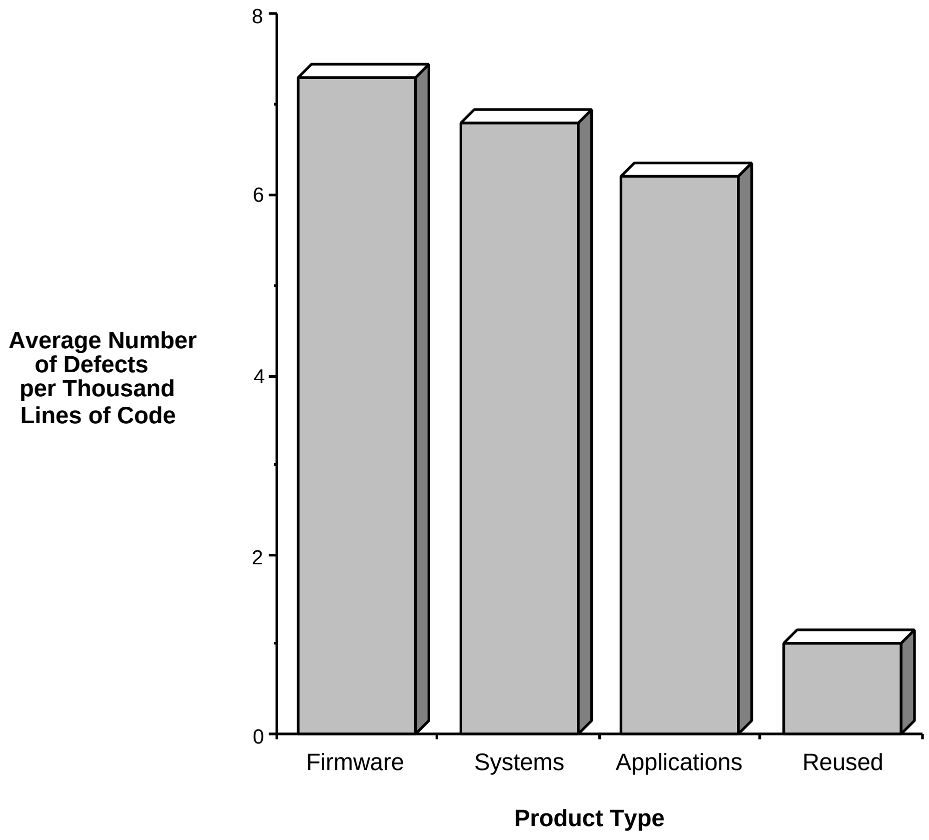


Figure D-10 Densities of Defects Discovered Before Products Were Released

In Figure D-11, we have a related example in which normalized quality measures are used to provide greater insight into the relationships between defect densities and productivity. To prepare this graph, the organization first ranked its projects in increasing order of productivity. (Useful measures of software productivity also depend on well-defined measures of software size.) The organization then plotted the observed defect densities against the productivity rankings. Although some scatter in the results was found, the figure shows a clear relationship between increasing quality (low defect densities) and increasing productivity.

The relationship in Figure D-1 may well have explainable causes. On the one hand, it may be taken as evidence that reducing defects improves productivity. On the other hand, it may simply be evidence that less complex products have both lower defects and higher productivities. If further investigation shows that the observed trend is not related to product complexity, information like this can be helpful in convincing organizations that one of the best ways to improve productivity is to focus on preventing defects. Otherwise, it can suggest the magnitude of the benefits that can be gained from reducing product complexities.

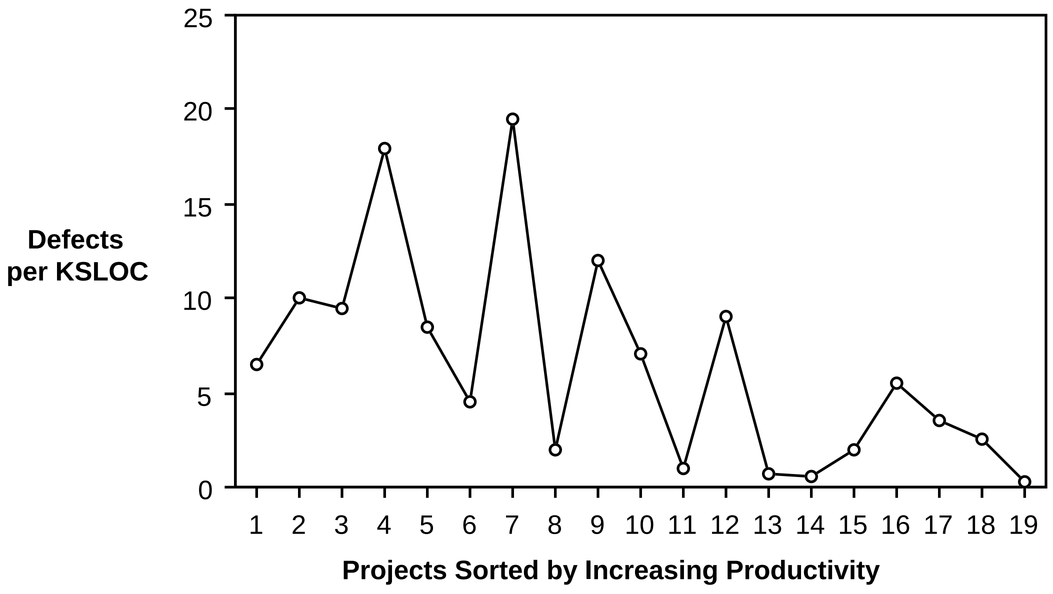


Figure D-11 Defect Densities for Firmware

# Appendix E: Forms for Reproduction

The following figures are repeated in this appendix in reproducible form. Figure numbers, page numbers, document footers, and sample entries have been removed, so that readers can copy and use the pages for their own purposes.

| | | Original Page Number |
|---|---|---|
| Figure 3-2 | Definition Checklist for Source Statement Counts | 16 |
| Figure 5-1 | Definition for Physical Source Lines of Code (SLOC) | 61 |
| Figure 5-2 | Definition for Logical Source Statements | 67 |
| Figure 5-4 | Data Spec A (Project Tracking) | 74 |
| Figure 5-9 | Data Spec B (Project Analysis) | 80 |
| Figure 5-10 | Data Spec C (Reuse Measurement) | 83 |
| Figure 5-11 | Data Spec B+C (Combined Specifications) | 86 |
| Figure 7-1 | Rules Form—Counting Physical Source Lines | 104 |
| Figure 7-2 | Rules Form—Counting Logical Source Statements | 105 |
| Figure 7-3 | Practices Used to Identify Inoperative Elements | 106 |
| Figure 8-1 | Recording Form for Data Spec A | 108 |
| Figure 8-2 | Recording Form for Data Spec B | 109 |
| Figure 8-3 | Recording Form for Data Spec C | 111 |
| Figure 8-4 | A Generalized Form for Recording Measurements of Source Code Size | 113 |
| Figure 9-1 | Summary Size Report | 116 |
| Figure 9-2 | Summary Reporting Form for Data Spec A (Project Tracking) | 119 |
| Figure 9-6 | Request for a Summary Report of Attribute Totals | 127 |

# Definition Checklist for Source Statement Counts

Definition name:________________________________ Date: __________

____________________________________________ Originator:__________

**Measurement unit:** **Physical source lines** ☐ **Logical source statements** ☐

| | **Statement type** **Definition** ☐ **Data array** ☐ | | **Includes** | **Excludes** |
|---|---|---|---|---|
| | *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | |
| 1 | Executable **Order of precedence ->** | 1 | | |
| 2 | Nonexecutable | | | |
| 3 | Declarations | 2 | | |
| 4 | Compiler directives | 3 | | |
| 5 | Comments | | | |
| 6 | On their own lines | 4 | | |
| 7 | On lines with source cod | 5 | | |
| 8 | Banners and nonblank spacers | 6 | | |
| 9 | Blank (empty) comments | 7 | | |
| 10 | Blank lines | 8 | | |
| 11 | | | | |
| 12 | | | | |

| | **How produced** **Definition** ☐ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|
| 1 | Programmed | | |
| 2 | Generated with source code generators | | |
| 3 | Converted with automated translators | | |
| 4 | Copied or reused without change | | |
| 5 | Modified | | |
| 6 | Removed | | |
| 7 | | | |
| 8 | | | |

| | **Origin** **Definition** ☐ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|
| 1 | New work: no prior existence | | |
| 2 | Prior work: taken or adapted from | | |
| 3 | A previous version, build, or release | | |
| 4 | Commercial, off-the-shelf software (COTS), other than libraries | | |
| 5 | Government furnished software (GFS), other than reuse libraries | | |
| 6 | Another product | | |
| 7 | A vendor-supplied language support library (unmodified) | | |
| 8 | A vendor-supplied operating system or utility (unmodified) | | |
| 9 | A local or modified language support library or operating system | | |
| 10 | Other commercial library | | |
| 11 | A reuse library (software designed for reus | | |
| 12 | Other software component or library | | |
| 13 | | | |
| 14 | | | |

| | **Usage** **Definition** ☐ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|
| 1 | In or as part of the primary product | | |
| 2 | External to or in support of the primary product | | |
| 3 | | | |

Definition name: ______________________________

______________________________

| | | | | |
|---|---|---|---|---|
| **Delivery** | **Definition** ☐ | **Data array** ☐ | **Includes** | **Excludes** |
| 1 Delivered | | | | |
| 2 Delivered as source | | | | |
| 3 Delivered in compiled or executable form, but not as source | | | | |
| 4 Not delivered | | | | |
| 5 Under configuration control | | | | |
| 6 Not under configuration control | | | | |
| 7 | | | | |
| **Functionality** | **Definition** ☐ | **Data array** ☐ | **Includes** | **Excludes** |
| 1 Operative | | | | |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | | | |
| 3 Functional (intentional dead code, reactivated for special purposes) | | | | |
| 4 Nonfunctional (unintentionally present) | | | | |
| 5 | | | | |
| 6 | | | | |
| **Replications** | **Definition** ☐ | **Data array** ☐ | **Includes** | **Excludes** |
| 1 Master source statements (originals) | | | | |
| 2 Physical replicates of master statements, stored in the master code | | | | |
| 3 Copies inserted, instantiated, or expanded when compiling or linking | | | | |
| 4 Postproduction replicates—as in distributed, redundant, or reparameterized systems | | | | |
| 5 | | | | |
| **Development status** | **Definition** ☐ | **Data array** ☐ | **Includes** | **Excludes** |
| *Each statement has one and only one status, usually that of its parent unit.* | | | | |
| 1 Estimated or planned | | | | |
| 2 Designed | | | | |
| 3 Coded | | | | |
| 4 Unit tests completed | | | | |
| 5 Integrated into components | | | | |
| 6 Test readiness review completed | | | | |
| 7 Software (CSCI) tests completed | | | | |
| 8 System tests completed | | | | |
| 9 | | | | |
| 10 | | | | |
| 11 | | | | |
| **Language** | **Definition** ☐ | **Data array** ☐ | **Includes** | **Excludes** |
| *List each source language on a separate line.* | | | | |
| 1 | | | | |
| 2 Job control languages | | | | |
| 3 | | | | |
| 4 Assembly languages | | | | |
| 5 | | | | |
| 6 Third generation languages | | | | |
| 7 | | | | |
| 8 Fourth generation languages | | | | |
| 9 | | | | |
| 10 Microcode | | | | |
| 11 | | | | |

| Definition name:________________________________ ________________________________ | | Includes | Excludes |
|---|---|---|---|
| **Clarifications (general)** **Listed elements are assigned to statement type –>** | | | |
| 1 Nulls, continues, and no-ops | | | |
| 2 Empty statements (e.g., “;;” and lone semicolons on separate lines) | | | |
| 3 Statements that instantiate generics | | | |
| 4 Begin…end and {…} pairs used as executable statements | | | |
| 5 Begin…end and {…} pairs that delimit (sub)program bodies | | | |
| 6 Logical expressions used as test conditions | | | |
| 7 Expression evaluations used as subprogram arguments | | | |
| 8 End symbols that terminate executable statements | | | |
| 9 End symbols that terminate declarations or (sub)program bod | | | |
| 10 Then, else, and otherwise symbols | | | |
| 11 Elseif statements | | | |
| 12 Keywords like procedure division, interface, and implementati | | | |
| 13 Labels (branching destinations) on lines by themselves | | | |
| 14 | | | |
| 15 | | | |
| 16 | | | |
| **Clarifications (language specific)** | | | |
| **Ada** | | | |
| 1 End symbols that terminate declarations or (sub)program bod | | | |
| 2 Block statements (e.g., begin…end) | | | |
| 3 With and use clauses | | | |
| 4 When (the keyword preceding executable statements) | | | |
| 5 Exception (the keyword, used as a frame header) | | | |
| 6 Pragmas | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **Assembly** | | | |
| 1 Macro calls | | | |
| 2 Macro expansions | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| **C and C++** | | | |
| 1 Null statement (e.g., “;” by itself to indicate an empty body) | | | |
| 2 Expression statements (expressions terminated by semicolons) | | | |
| 3 Expressions separated by semicolons, as in a "for" statement | | | |
| 4 Block statements (e.g., {…} with no terminating semicolon) | | | |
| 5 “{”, “}”, or “};” on a line by itself when part of a declaration | | | |
| 6 “{” or “}” on line by itself when part of an executable statement | | | |
| 7 Conditionally compiled statements (#if, #ifdef, #ifndef) | | | |
| 8 Preprocessor statements other than #if, #ifdef, and #ifndef | | | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |

| Definition name:______________________________ ______________________________ | | Includes | Excludes |
|---|---|---|---|
| **CMS-2** **Listed elements are assigned to statement type –>** | | | |
| 1 Keywords like SYS-PROC and SYS-DD | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **COBOL** | | | |
| 1 “PROCEDURE DIVISION”, “END DECLARATIVES”, etc. | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **FORTRAN** | | | |
| 1 END statements | | | |
| 2 Format statements | | | |
| 3 Entry statements | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| **JOVIAL** | | | |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| **Pascal** | | | |
| 1 Executable statements not terminated by semicolons | | | |
| 2 Keywords like INTERFACE and IMPLEMENTATION | | | |
| 3 FORWARD declarations | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |

| Definition name:______________________ | | Includes | Excludes |
|---|---|---|---|
| Listed elements are assigned to statement type –> | | | |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |

**Summary of Statement Types**

**Executable statements**

Executable statements cause runtime actions. They may be simple statements such as assignments, goto's, procedure calls, macro calls, returns, breaks, exits, stops, continues, nulls no-ops, empty statements, and FORTRAN's END. Or they may be structured or compound statements, such as conditional statements, repetitive statements, and "with" statements. Languages like Ada, C, C++, and Pascal have block statements [begin…end and {…}] that are classified as executable when used where other executable statements would be permitted. C and C++ define expressions as executable statements when they terminate with a semicolon, and C++ has a <declaration> statement that is executable.

**Declarations**

Declarations are nonexecutable program elements that affect an assembler's or compiler's interpretation of other program elements. They are used to name, define, and initialize; to specify internal and external interfaces; to assign ranges for bounds checking; and to identify and bound modules and sections of code. Examples include declarations of names, numbers, constants, objects, types, subtypes, programs, subprograms, tasks, exceptions, packages, generics, macros, and deferred constants. Declarations also include renaming declarations, us clauses, and declarations that instantiate generics. Mandatory begin…end and {…} symbols th delimit bodies of programs and subprograms are integral parts of program and subprogram declarations. Language superstructure elements that establish boundaries for different sections of source code are also declarations. Examples include terms such as PROCEDURE DIVISION DATA DIVISION, DECLARATIVES, END DECLARATIVES, INTERFACE, IMPLEMENTATION, SYS-PROC, and SYS-DD. Declarations, in general, are never required by language specifications to initiate runtime actions, although some languages permit compile implement them that way.

**Compiler Directives**

Compiler directives instruct compilers, preprocessors, or translators (but not runtime systems) to perform special actions. Some, such as Ada's pragma and COBOL's COPY, REPLACE, an USE, are integral parts of the source language. In other languages like C and C++, special symbols like # are used along with standardized keywords to direct preprocessor or compiler actions. Still other languages rely on nonstandardized methods supplied by compiler vendors. In these languages, directives are often designated by special symbols such as #, $, and {$}.

# Definition Checklist for Source Statement Counts

Definition name: *Physical Source Lines of Code (basic definition)*

Date: *8/7/92*

Originator *SEI*

| | | |
|---|---|---|
| **Measurement unit:** | **Physical source lines** | ✓ |
| | **Logical source statements** | |

| **Statement type** **Definition** [✓] **Data array** [ ] | | **Includes** | **Excludes** |
|---|---|---|---|
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | ▨ | ▨ |
| 1 Executable **Order of precedence ->** | 1 | ✓ | |
| 2 Nonexecutable | | ▨ | ▨ |
| 3 Declarations | 2 | ✓ | |
| 4 Compiler directives | 3 | ✓ | |
| 5 Comments | | ▨ | ▨ |
| 6 On their own lines | 4 | | ✓ |
| 7 On lines with source cod | 5 | | ✓ |
| 8 Banners and nonblank spacers | 6 | | ✓ |
| 9 Blank (empty) comments | 7 | | ✓ |
| 10 Blank lines | 8 | | ✓ |
| 11 | | | |
| 12 | | | |

| **How produced** **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 Programmed | ✓ | |
| 2 Generated with source code generators | ✓ | |
| 3 Converted with automated translators | ✓ | |
| 4 Copied or reused without change | ✓ | |
| 5 Modified | ✓ | |
| 6 Removed | | ✓ |
| 7 | | |
| 8 | | |

| **Origin** **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 New work: no prior existence | ✓ | |
| 2 Prior work: taken or adapted from | ▨ | ▨ |
| 3 A previous version, build, or release | ✓ | |
| 4 Commercial, off-the-shelf software (COTS), other than libraries | ✓ | |
| 5 Government furnished software (GFS), other than reuse libraries | ✓ | |
| 6 Another product | ✓ | |
| 7 A vendor-supplied language support library (unmodified) | | ✓ |
| 8 A vendor-supplied operating system or utility (unmodified) | | ✓ |
| 9 A local or modified language support library or operating system | ✓ | |
| 10 Other commercial library | ✓ | |
| 11 A reuse library (software designed for reus | ✓ | |
| 12 Other software component or library | ✓ | |
| 13 | | |
| 14 | | |

| **Usage** **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 In or as part of the primary product | ✓ | |
| 2 External to or in support of the primary product | | ✓ |
| 3 | | |

Definition name: ***Physical Source Lines of Code***
***(basic definition)***

| Delivery | Definition [✓] Data array [ ] | Includes | Excludes |
|---|---|---|---|
| 1 | Delivered | | |
| 2 | Delivered as source | ✓ | |
| 3 | Delivered in compiled or executable form, but not as source | ✓ | |
| 4 | Not delivered | | |
| 5 | Under configuration control | | ✓ |
| 6 | Not under configuration control | | ✓ |
| 7 | | | |

| Functionality | Definition [✓] Data array [ ] | Includes | Excludes |
|---|---|---|---|
| 1 | Operative | ✓ | |
| 2 | Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | |
| 3 | Functional (intentional dead code, reactivated for special purposes) | ✓ | |
| 4 | Nonfunctional (unintentionally present) | | ✓ |
| 5 | | | |
| 6 | | | |

| Replications | Definition [✓] Data array [ ] | Includes | Excludes |
|---|---|---|---|
| 1 | Master source statements (originals) | ✓ | |
| 2 | Physical replicates of master statements, stored in the master code | ✓ | |
| 3 | Copies inserted, instantiated, or expanded when compiling or linking | | ✓ |
| 4 | Postproduction replicates—as in distributed, redundant, or reparameterized systems | | ✓ |
| 5 | | | |

| Development status | Definition [✓] Data array [ ] | Includes | Excludes |
|---|---|---|---|
| | *Each statement has one and only one status, usually that of its parent unit.* | | |
| 1 | Estimated or planned | | ✓ |
| 2 | Designed | | ✓ |
| 3 | Coded | | ✓ |
| 4 | Unit tests completed | | ✓ |
| 5 | Integrated into components | | ✓ |
| 6 | Test readiness review completed | | ✓ |
| 7 | Software (CSCI) tests completed | | ✓ |
| 8 | System tests completed | ✓ | |
| 9 | | | |
| 10 | | | |
| 11 | | | |

| Language | Definition [ ] Data array [✓] | Includes | Excludes |
|---|---|---|---|
| | *List each source language on a separate line.* | | |
| 1 | ***Separate totals for each language*** | ✓ | |
| 2 | Job control languages | | |
| 3 | | | |
| 4 | Assembly languages | | |
| 5 | | | |
| 6 | Third generation languages | | |
| 7 | | | |
| 8 | Fourth generation languages | | |
| 9 | | | |
| 10 | Microcode | | |
| 11 | | | |

| Definition name: ***Physical Source Lines of Code (basic definition)*** | | Includes | Excludes |
|---|---|---|---|
| **Clarifications (general)** **Listed elements are assigned to statement type –>** | | | |
| 1 Nulls, continues, and no-ops | *1* | ✔ | |
| 2 Empty statements (e.g., “;;” and lone semicolons on separate lines) | *1* | ✔ | |
| 3 Statements that instantiate generics | *3* | ✔ | |
| 4 Begin…end and {…} pairs used as executable statements | *1* | ✔ | |
| 5 Begin…end and {…} pairs that delimit (sub)program bodies | *3* | ✔ | |
| 6 Logical expressions used as test conditions | *1* | ✔ | |
| 7 Expression evaluations used as subprogram arguments | *1* | ✔ | |
| 8 End symbols that terminate executable statements | *1* | ✔ | |
| 9 End symbols that terminate declarations or (sub)program bod | *3* | ✔ | |
| 10 Then, else, and otherwise symbols | *1* | ✔ | |
| 11 Elseif statements | *1* | ✔ | |
| 12 Keywords like procedure division, interface, and implementati | *3* | ✔ | |
| 13 Labels (branching destinations) on lines by themselves | *1* | ✔ | |
| 14 | | | |
| 15 | | | |
| 16 | | | |
| **Clarifications (language specific)** | | | |
| **Ada** | | | |
| 1 End symbols that terminate declarations or (sub)program bod | *3* | ✔ | |
| 2 Block statements (e.g., begin…end) | *1* | ✔ | |
| 3 With and use clauses | *3* | ✔ | |
| 4 When (the keyword preceding executable statements) | *1* | ✔ | |
| 5 Exception (the keyword, used as a frame header) | *3* | ✔ | |
| 6 Pragmas | *4* | ✔ | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **Assembly** | | | |
| 1 Macro calls | *1* | ✔ | |
| 2 Macro expansions | | | ✔ |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| **C and C++** | | | |
| 1 Null statement (e.g., “;” by itself to indicate an empty body) | *1* | ✔ | |
| 2 Expression statements (expressions terminated by semicolons) | *1* | ✔ | |
| 3 Expressions separated by semicolons, as in a "for" statement | *1* | ✔ | |
| 4 Block statements (e.g., {…} with no terminating semicolon) | *1* | ✔ | |
| 5 “{”, “}”, or “};” on a line by itself when part of a declaration | *3* | ✔ | |
| 6 “{” or “}” on line by itself when part of an executable statement | *1* | ✔ | |
| 7 Conditionally compiled statements (#if, #ifdef, #ifndef) | *4* | ✔ | |
| 8 Preprocessor statements other than #if, #ifdef, and #ifndef | *4* | ✔ | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |

| Definition name: ***Physical Source Lines of Code (basic definition)*** | | Includes | Excludes |
|---|---|---|---|
| **CMS-2** **Listed elements are assigned to** | | | |
| 1 Keywords like SYS-PROC and SYS-DD **statement type –>** | ***3*** | ✔ | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **COBOL** | | | |
| 1 “PROCEDURE DIVISION”, “END DECLARATIVES”, etc. | ***3*** | ✔ | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **FORTRAN** | | | |
| 1 END statements | ***1*** | ✔ | |
| 2 Format statements | ***3*** | ✔ | |
| 3 Entry statements | ***3*** | ✔ | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| **JOVIAL** | | | |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| **Pascal** | | | |
| 1 Executable statements not terminated by semicolons | ***1*** | ✔ | |
| 2 Keywords like INTERFACE and IMPLEMENTATION | ***3*** | ✔ | |
| 3 FORWARD declarations | ***3*** | ✔ | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |

| Definition name: ***Physical Source Lines of Code*** ***(basic definition)*** | | **Includes** | **Excludes** |
|---|---|---|---|
| **Listed elements are assigned to statement type –>** | | | |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |

**Summary of Statement Types**

**Executable statements**

Executable statements cause runtime actions. They may be simple statements such as assignments, goto's, procedure calls, macro calls, returns, breaks, exits, stops, continues, nulls no-ops, empty statements, and FORTRAN's END. Or they may be structured or compound statements, such as conditional statements, repetitive statements, and "with" statements. Languages like Ada, C, C++, and Pascal have block statements [begin…end and {…}] that are classified as executable when used where other executable statements would be permitted. C and C++ define expressions as executable statements when they terminate with a semicolon, and C++ has a <declaration> statement that is executable.

**Declarations**

Declarations are nonexecutable program elements that affect an assembler's or compiler's interpretation of other program elements. They are used to name, define, and initialize; to specify internal and external interfaces; to assign ranges for bounds checking; and to identify and bound modules and sections of code. Examples include declarations of names, numbers, constants, objects, types, subtypes, programs, subprograms, tasks, exceptions, packages, generics, macros, and deferred constants. Declarations also include renaming declarations, us clauses, and declarations that instantiate generics. Mandatory begin…end and {…} symbols th delimit bodies of programs and subprograms are integral parts of program and subprogram declarations. Language superstructure elements that establish boundaries for different sections of source code are also declarations. Examples include terms such as PROCEDURE DIVISION DATA DIVISION, DECLARATIVES, END DECLARATIVES, INTERFACE, IMPLEMENTATION, SYS-PROC, and SYS-DD. Declarations, in general, are never required by language specifications to initiate runtime actions, although some languages permit compile implement them that way.

**Compiler Directives**

Compiler directives instruct compilers, preprocessors, or translators (but not runtime systems) to perform special actions. Some, such as Ada's pragma and COBOL's COPY, REPLACE, an USE, are integral parts of the source language. In other languages like C and C++, special symbols like # are used along with standardized keywords to direct preprocessor or compiler actions. Still other languages rely on nonstandardized methods supplied by compiler vendors. In these languages, directives are often designated by special symbols such as #, $, and {$}.

# Definition Checklist for Source Statement Counts

Definition name: ***Logical Source Statements*** Date: ***8/7/92***

***(basic definition)*** Originator ***SEI***

| | | |
|---|---|---|
| **Measurement unit:** | **Physical source lines** | ☐ |
| | **Logical source statements** | ✓ |

| **Statement type** | **Definition** ✓ **Data array** ☐ | | **Includes** | **Excludes** |
|---|---|---|---|---|
| | *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | |
| 1 | Executable | **Order of precedence ->** 1 | ✓ | |
| 2 | Nonexecutable | | | |
| 3 | Declarations | 2 | ✓ | |
| 4 | Compiler directives | 3 | ✓ | |
| 5 | Comments | | | |
| 6 | On their own lines | 4 | | ✓ |
| 7 | On lines with source cod | 5 | | ✓ |
| 8 | Banners and nonblank spacers | 6 | | ✓ |
| 9 | Blank (empty) comments | 7 | | ✓ |
| 10 | Blank lines | 8 | | ✓ |
| 11 | | | | |
| 12 | | | | |

| **How produced** | **Definition** ✓ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|
| 1 | Programmed | ✓ | |
| 2 | Generated with source code generators | ✓ | |
| 3 | Converted with automated translators | ✓ | |
| 4 | Copied or reused without change | ✓ | |
| 5 | Modified | ✓ | |
| 6 | Removed | | ✓ |
| 7 | | | |
| 8 | | | |

| **Origin** | **Definition** ✓ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|
| 1 | New work: no prior existence | ✓ | |
| 2 | Prior work: taken or adapted from | | |
| 3 | A previous version, build, or release | ✓ | |
| 4 | Commercial, off-the-shelf software (COTS), other than libraries | ✓ | |
| 5 | Government furnished software (GFS), other than reuse libraries | ✓ | |
| 6 | Another product | ✓ | |
| 7 | A vendor-supplied language support library (unmodified) | | ✓ |
| 8 | A vendor-supplied operating system or utility (unmodified) | | ✓ |
| 9 | A local or modified language support library or operating system | ✓ | |
| 10 | Other commercial library | ✓ | |
| 11 | A reuse library (software designed for reus | ✓ | |
| 12 | Other software component or library | ✓ | |
| 13 | | | |
| 14 | | | |

| **Usage** | **Definition** ✓ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|
| 1 | In or as part of the primary product | ✓ | |
| 2 | External to or in support of the primary product | | ✓ |
| 3 | | | |

Definition name: ***Logical Source Statements***
***(basic definition)***

| Delivery — Definition ✓ Data array | Includes | Excludes |
|---|---|---|
| 1 Delivered | | |
| 2 Delivered as source | ✓ | |
| 3 Delivered in compiled or executable form, but not as source | ✓ | |
| 4 Not delivered | | |
| 5 Under configuration control | | ✓ |
| 6 Not under configuration control | | ✓ |
| 7 | | |

| Functionality — Definition ✓ Data array | Includes | Excludes |
|---|---|---|
| 1 Operative | ✓ | |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | |
| 3 Functional (intentional dead code, reactivated for special purposes) | ✓ | |
| 4 Nonfunctional (unintentionally present) | | ✓ |
| 5 | | |
| 6 | | |

| Replications — Definition ✓ Data array | Includes | Excludes |
|---|---|---|
| 1 Master source statements (originals) | ✓ | |
| 2 Physical replicates of master statements, stored in the master code | ✓ | |
| 3 Copies inserted, instantiated, or expanded when compiling or linking | | ✓ |
| 4 Postproduction replicates—as in distributed, redundant, or reparameterized systems | | ✓ |
| 5 | | |

| Development status — Definition ✓ Data array | Includes | Excludes |
|---|---|---|
| *Each statement has one and only one status, usually that of its parent unit.* | | |
| 1 Estimated or planned | | ✓ |
| 2 Designed | | ✓ |
| 3 Coded | | ✓ |
| 4 Unit tests completed | | ✓ |
| 5 Integrated into components | | ✓ |
| 6 Test readiness review completed | | ✓ |
| 7 Software (CSCI) tests completed | | ✓ |
| 8 System tests completed | ✓ | |
| 9 | | |
| 10 | | |
| 11 | | |

| Language — Definition Data array ✓ | Includes | Excludes |
|---|---|---|
| *List each source language on a separate line.* | | |
| 1 ***Separate totals for each language*** | ✓ | |
| 2 Job control languages | | |
| 3 | | |
| 4 Assembly languages | | |
| 5 | | |
| 6 Third generation languages | | |
| 7 | | |
| 8 Fourth generation languages | | |
| 9 | | |
| 10 Microcode | | |
| 11 | | |

| Definition name: ***Logical Source Statements (basic definition)*** | | Includes | Excludes |
|---|---|---|---|
| **Clarifications (general)** **Listed elements are assigned to statement type –>** | | | |
| 1 Nulls, continues, and no-ops | *1* | ✔ | |
| 2 Empty statements (e.g., “;;” and lone semicolons on separate lines) | | | ✔ |
| 3 Statements that instantiate generics | *3* | ✔ | |
| 4 Begin…end and {…} pairs used as executable statements | *1* | ✔ | |
| 5 Begin…end and {…} pairs that delimit (sub)program bodies | | | ✔ |
| 6 Logical expressions used as test conditions | | | ✔ |
| 7 Expression evaluations used as subprogram arguments | | | ✔ |
| 8 End symbols that terminate executable statements | | | ✔ |
| 9 End symbols that terminate declarations or (sub)program bod | | | ✔ |
| 10 Then, else, and otherwise symbols | | | ✔ |
| 11 Elseif statements | *1* | ✔ | |
| 12 Keywords like procedure division, interface, and implementati | *3* | ✔ | |
| 13 Labels (branching destinations) on lines by themselves | | | ✔ |
| 14 | | | |
| 15 | | | |
| 16 | | | |
| **Clarifications (language specific)** | | | |
| **Ada** | | | |
| 1 End symbols that terminate declarations or (sub)program bod | | | ✔ |
| 2 Block statements (e.g., begin…end) | *1* | ✔ | |
| 3 With and use clauses | *3* | ✔ | |
| 4 When (the keyword preceding executable statements) | | | ✔ |
| 5 Exception (the keyword, used as a frame header) | *3* | ✔ | |
| 6 Pragmas | *4* | ✔ | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **Assembly** | | | |
| 1 Macro calls | *1* | ✔ | |
| 2 Macro expansions | | | ✔ |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| **C and C++** | | | |
| 1 Null statement (e.g., “;” by itself to indicate an empty body) | | | ✔ |
| 2 Expression statements (expressions terminated by semicolons) | *1* | ✔ | |
| 3 Expressions separated by semicolons, as in a "for" statement | *1* | ✔ | |
| 4 Block statements (e.g., {…} with no terminating semicolon) | *1* | ✔ | |
| 5 “{”, “}”, or “};” on a line by itself when part of a declaration | | | ✔ |
| 6 “{” or “}” on line by itself when part of an executable statement | | | ✔ |
| 7 Conditionally compiled statements (#if, #ifdef, #ifndef) | *4* | ✔ | |
| 8 Preprocessor statements other than #if, #ifdef, and #ifndef | *4* | ✔ | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |

| Definition name: ***Logical Source Statements (basic definition)*** | | Includes | Excludes |
|---|---|---|---|
| **CMS-2** **Listed elements are assigned to** | | | |
| 1 Keywords like SYS-PROC and SYS-DD **statement type –>** | ***3*** | ✔ | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **COBOL** | | | |
| 1 "PROCEDURE DIVISION", "END DECLARATIVES", etc. | ***3*** | ✔ | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| **FORTRAN** | | | |
| 1 END statements | ***1*** | ✔ | |
| 2 Format statements | ***3*** | ✔ | |
| 3 Entry statements | ***3*** | ✔ | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| **JOVIAL** | | | |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| **Pascal** | | | |
| 1 Executable statements not terminated by semicolons | ***1*** | ✔ | |
| 2 Keywords like INTERFACE and IMPLEMENTATION | ***3*** | ✔ | |
| 3 FORWARD declarations | ***3*** | ✔ | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |

| Definition name: ***Logical Source Statements (basic definition)*** | | Includes | Excludes |
|---|---|---|---|
| **Listed elements are assigned to statement type –>** | | ▨ | ▨ |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| 12 | | | |

**Summary of Statement Types**

**Executable statements**

Executable statements cause runtime actions.  They may be simple statements such as assignments, goto's, procedure calls, macro calls, returns, breaks, exits, stops, continues, nulls no-ops, empty statements, and FORTRAN's END.  Or they may be structured or compound statements, such as conditional statements, repetitive statements, and "with" statements. Languages like Ada, C, C++, and Pascal have block statements [begin…end and {…}] that are classified as executable when used where other executable statements would be permitted.  C and C++ define expressions as executable statements when they terminate with a semicolon, and C++ has a <declaration> statement that is executable.

**Declarations**

Declarations are nonexecutable program elements that affect an assembler's or compiler's interpretation of other program elements.  They are used to name, define, and initialize; to specify internal and external interfaces; to assign ranges for bounds checking; and to identify and bound modules and sections of code. Examples include declarations of names, numbers, constants, objects, types, subtypes, programs, subprograms, tasks, exceptions, packages, generics, macros, and deferred constants.  Declarations also include renaming declarations, us clauses, and declarations that instantiate generics.  Mandatory begin…end and {…} symbols th delimit bodies of programs and subprograms are integral parts of program and subprogram declarations. Language superstructure elements that establish boundaries for different section of source code are also declarations.  Examples include terms such as PROCEDURE DIVISIO DATA DIVISION, DECLARATIVES, END DECLARATIVES, INTERFACE, IMPLEMENTATION, SYS-PROC, and SYS-DD.  Declarations, in general, are never required by language specifications to initiate runtime actions, although some languages permit compile implement them that way.

**Compiler Directives**

Compiler directives instruct compilers, preprocessors, or translators (but not runtime systems) to perform special actions.  Some, such as Ada's pragma and COBOL's COPY, REPLACE, an USE, are integral parts of the source language.  In other languages like C and C++, special symbols like # are used along with standardized keywords to direct preprocessor or compiler actions.  Still other languages rely on nonstandardized methods supplied by compiler vendors. In these languages, directives are often designated by special symbols such as #, $, and {$}.

# Definition Checklist for Source Statement Counts

Definition name: ***Data Spec A: Project Tracking Example*** Date: ***8/7/92***

***(for tracking status vs. how produced)*** Originator ***SEI***

**Measurement unit:** **Physical source lines** [ ] **Logical source statements** [ ]

| | | Order of precedence | Includes | Excludes |
|---|---|---|---|---|
| **Statement type** | **Definition** [✓] **Data array** [ ] | | **Includes** | **Excludes** |
| | *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | |
| 1 | Executable **Order of precedence ->** | 1 | ✓ | |
| 2 | Nonexecutable | | | |
| 3 | Declarations | 2 | ✓ | |
| 4 | Compiler directives | 3 | ✓ | |
| 5 | Comments | | | |
| 6 | On their own lines | 4 | | ✓ |
| 7 | On lines with source cod | 5 | | ✓ |
| 8 | Banners and nonblank spacers | 6 | | ✓ |
| 9 | Blank (empty) comments | 7 | | ✓ |
| 10 | Blank lines | 8 | | ✓ |
| 11 | | | | |
| 12 | | | | |
| **How produced** | **Definition** [ ] **Data array** [✓] | | **Includes** | **Excludes** |
| 1 | Programmed | | ✓ | |
| 2 | Generated with source code generators | | ✓ | |
| 3 | Converted with automated translators | | ✓ | |
| 4 | Copied or reused without change | | ✓ | |
| 5 | Modified | | ✓ | |
| 6 | Removed | | ✓ | |
| 7 | | | | |
| 8 | | | | |
| **Origin** | **Definition** [✓] **Data array** [ ] | | **Includes** | **Excludes** |
| 1 | New work: no prior existence | | ✓ | |
| 2 | Prior work: taken or adapted from | | | |
| 3 | A previous version, build, or release | | ✓ | |
| 4 | Commercial, off-the-shelf software (COTS), other than libraries | | ✓ | |
| 5 | Government furnished software (GFS), other than reuse libraries | | ✓ | |
| 6 | Another product | | ✓ | |
| 7 | A vendor-supplied language support library (unmodified) | | | ✓ |
| 8 | A vendor-supplied operating system or utility (unmodified) | | | ✓ |
| 9 | A local or modified language support library or operating system | | ✓ | |
| 10 | Other commercial library | | ✓ | |
| 11 | A reuse library (software designed for reus | | ✓ | |
| 12 | Other software component or library | | ✓ | |
| 13 | | | | |
| 14 | | | | |
| **Usage** | **Definition** [✓] **Data array** [ ] | | **Includes** | **Excludes** |
| 1 | In or as part of the primary product | | ✓ | |
| 2 | External to or in support of the primary product | | | ✓ |
| 3 | | | | |

Definition name: ***Data Spec A: Project Tracking Example***
***(for tracking status vs. how produced)***

| **Delivery** | **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
|---|---|---|---|
| 1 | Delivered | | |
| 2 | Delivered as source | ✓ | |
| 3 | Delivered in compiled or executable form, but not as source | ✓ | |
| 4 | Not delivered | | |
| 5 | Under configuration control | | ✓ |
| 6 | Not under configuration control | | ✓ |
| 7 | | | |

| **Functionality** | **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
|---|---|---|---|
| 1 | Operative | ✓ | |
| 2 | Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | |
| 3 | Functional (intentional dead code, reactivated for special purposes) | ✓ | |
| 4 | Nonfunctional (unintentionally present) | | ✓ |
| 5 | | | |
| 6 | | | |

| **Replications** | **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
|---|---|---|---|
| 1 | Master source statements (originals) | ✓ | |
| 2 | Physical replicates of master statements, stored in the master code | ✓ | |
| 3 | Copies inserted, instantiated, or expanded when compiling or linking | | ✓ |
| 4 | Postproduction replicates—as in distributed, redundant, or reparameterized systems | | ✓ |
| 5 | | | |

| **Development status** | **Definition** [ ] **Data array** [✓] | **Includes** | **Excludes** |
|---|---|---|---|
| | *Each statement has one and only one status, usually that of its parent unit.* | | |
| 1 | Estimated or planned | | ✓ |
| 2 | Designed | | ✓ |
| 3 | Coded | ✓ | |
| 4 | Unit tests completed | ✓ | |
| 5 | Integrated into components | ✓ | |
| 6 | Test readiness review completed | ✓ | |
| 7 | Software (CSCI) tests completed | ✓ | |
| 8 | System tests completed | ✓ | |
| 9 | | | |
| 10 | | | |
| 11 | | | |

| **Language** | **Definition** [ ] **Data array** [✓] | | **Includes** | **Excludes** |
|---|---|---|---|---|
| | *List each source language on a separate line.* | | | |
| 1 | | ***Separate totals for each language*** | ✓ | |
| 2 | Job control languages | | | |
| 3 | | | | |
| 4 | Assembly languages | | | |
| 5 | | | | |
| 6 | Third generation languages | | | |
| 7 | | | | |
| 8 | Fourth generation languages | | | |
| 9 | | | | |
| 10 | Microcode | | | |
| 11 | | | | |

# Definition Checklist for Source Statement Counts

Definition name: ***Data Spec B: Project Analysis Example*** Date: ***8/7/92***

***(end-of-project data used to improve future estimates)*** Originator ***SEI***

**Measurement unit:** **Physical source lines** [ ] **Logical source statements** [ ]

| **Statement type** Definition [ ] Data array [✓] | Order of precedence | **Includes** | **Excludes** |
|---|---|---|---|
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | |
| 1 Executable | 1 | ✓ | |
| 2 Nonexecutable | | | |
| 3 Declarations | 2 | ✓ | |
| 4 Compiler directives | 3 | ✓ | |
| 5 Comments | | | |
| 6 On their own lines | 4 | ✓ | |
| 7 On lines with source cod | 5 | ✓ | |
| 8 Banners and nonblank spacers | 6 | | ✓ |
| 9 Blank (empty) comments | 7 | | ✓ |
| 10 Blank lines | 8 | | ✓ |
| 11 | | | |
| 12 | | | |

| **How produced** Definition [ ] Data array [✓] | **Includes** | **Excludes** |
|---|---|---|
| 1 Programmed | ✓ | |
| 2 Generated with source code generators | ✓ | |
| 3 Converted with automated translators | ✓ | |
| 4 Copied or reused without change | ✓ | |
| 5 Modified | ✓ | |
| 6 Removed | ✓ | |
| 7 | | |
| 8 | | |

| **Origin** Definition [✓] Data array [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 New work: no prior existence | ✓ | |
| 2 Prior work: taken or adapted from | | |
| 3 A previous version, build, or release | ✓ | |
| 4 Commercial, off-the-shelf software (COTS), other than libraries | ✓ | |
| 5 Government furnished software (GFS), other than reuse libraries | ✓ | |
| 6 Another product | ✓ | |
| 7 A vendor-supplied language support library (unmodified) | | ✓ |
| 8 A vendor-supplied operating system or utility (unmodified) | | ✓ |
| 9 A local or modified language support library or operating system | ✓ | |
| 10 Other commercial library | ✓ | |
| 11 A reuse library (software designed for reus | ✓ | |
| 12 Other software component or library | ✓ | |
| 13 | | |
| 14 | | |

| **Usage** Definition [✓] Data array [ ] | **Includes** | **Excludes** |
|---|---|---|
| 1 In or as part of the primary product | ✓ | |
| 2 External to or in support of the primary product | | ✓ |
| 3 | | |

Definition name: ***Data Spec B: Project Analysis Example (end-of-project data used to improve future estimates)***

| **Delivery** | **Definition** ✓ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|
| 1 | Delivered | | |
| 2 | Delivered as source | ✓ | |
| 3 | Delivered in compiled or executable form, but not as source | ✓ | |
| 4 | Not delivered | | |
| 5 | Under configuration control | | ✓ |
| 6 | Not under configuration control | | ✓ |
| 7 | | | |

| **Functionality** | **Definition** ✓ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|
| 1 | Operative | ✓ | |
| 2 | Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | |
| 3 | Functional (intentional dead code, reactivated for special purposes) | ✓ | |
| 4 | Nonfunctional (unintentionally present) | | ✓ |
| 5 | | | |
| 6 | | | |

| **Replications** | **Definition** ✓ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|
| 1 | Master source statements (originals) | ✓ | |
| 2 | Physical replicates of master statements, stored in the master code | ✓ | |
| 3 | Copies inserted, instantiated, or expanded when compiling or linking | | ✓ |
| 4 | Postproduction replicates—as in distributed, redundant, or reparameterized systems | | ✓ |
| 5 | | | |

| **Development status** | **Definition** ✓ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|---|
| | *Each statement has one and only one status, usually that of its parent unit.* | | |
| 1 | Estimated or planned | | ✓ |
| 2 | Designed | | ✓ |
| 3 | Coded | | ✓ |
| 4 | Unit tests completed | | ✓ |
| 5 | Integrated into components | | ✓ |
| 6 | Test readiness review completed | | ✓ |
| 7 | Software (CSCI) tests completed | | ✓ |
| 8 | System tests completed | ✓ | |
| 9 | | | |
| 10 | | | |
| 11 | | | |

| **Language** | **Definition** ☐ **Data array** ✓ | **Includes** | **Excludes** |
|---|---|---|---|
| | *List each source language on a separate line.* | | |
| 1 | ***Separate totals for each language*** | ✓ | |
| 2 | Job control languages | | |
| 3 | | | |
| 4 | Assembly languages | | |
| 5 | | | |
| 6 | Third generation languages | | |
| 7 | | | |
| 8 | Fourth generation languages | | |
| 9 | | | |
| 10 | Microcode | | |
| 11 | | | |

# Definition Checklist for Source Statement Counts

Definition name: ***Data Spec C: Reuse Measurement Example*** Date: ***8/7/92***

Originator ***SEI***

**Measurement unit:** **Physical source lines** [ ] **Logical source statements** [ ]

| **Statement type** | **Definition** [✓] **Data array** [ ] | | **Includes** | **Excludes** |
|---|---|---|---|---|
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | | |
| 1 Executable | **Order of precedence ->** | 1 | ✓ | |
| 2 Nonexecutable | | | | |
| 3 Declarations | | 2 | ✓ | |
| 4 Compiler directives | | 3 | ✓ | |
| 5 Comments | | | | |
| 6 On their own lines | | 4 | | ✓ |
| 7 On lines with source cod | | 5 | | ✓ |
| 8 Banners and nonblank spacers | | 6 | | ✓ |
| 9 Blank (empty) comments | | 7 | | ✓ |
| 10 Blank lines | | 8 | | ✓ |
| 11 | | | | |
| 12 | | | | |

| **How produced** | **Definition** [ ] **Data array** [✓] | **Includes** | **Excludes** |
|---|---|---|---|
| 1 Programmed | | ✓ | |
| 2 Generated with source code generators | | ✓ | |
| 3 Converted with automated translators | | ✓ | |
| 4 Copied or reused without change | | ✓ | |
| 5 Modified | | ✓ | |
| 6 Removed | | ✓ | |
| 7 | | | |
| 8 | | | |

| **Origin** | **Definition** [ ] **Data array** [✓] | **Includes** | **Excludes** |
|---|---|---|---|
| 1 New work: no prior existence | | ✓ | |
| 2 Prior work: taken or adapted from | | | |
| 3 A previous version, build, or release | | ✓ | |
| 4 Commercial, off-the-shelf software (COTS), other than libraries | | ✓ | |
| 5 Government furnished software (GFS), other than reuse libraries | | ✓ | |
| 6 Another product | | ✓ | |
| 7 A vendor-supplied language support library (unmodified) | | | ✓ |
| 8 A vendor-supplied operating system or utility (unmodified) | | | ✓ |
| 9 A local or modified language support library or operating system | | ✓ | |
| 10 Other commercial library | | ✓ | |
| 11 A reuse library (software designed for reus | | ✓ | |
| 12 Other software component or library | | ✓ | |
| 13 | | | |
| 14 | | | |

| **Usage** | **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
|---|---|---|---|
| 1 In or as part of the primary product | | ✓ | |
| 2 External to or in support of the primary product | | | ✓ |
| 3 | | | |

Definition name: ***Data Spec C: Reuse Measurement Example***

| Delivery | Definition [✓] | Data array [ ] | Includes | Excludes |
|---|---|---|---|---|
| 1 Delivered | | | | |
| 2 Delivered as source | | | ✓ | |
| 3 Delivered in compiled or executable form, but not as source | | | ✓ | |
| 4 Not delivered | | | | |
| 5 Under configuration control | | | | ✓ |
| 6 Not under configuration control | | | | ✓ |
| 7 | | | | |

| Functionality | Definition [✓] | Data array [ ] | Includes | Excludes |
|---|---|---|---|---|
| 1 Operative | | | ✓ | |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | | | |
| 3 Functional (intentional dead code, reactivated for special purposes) | | | ✓ | |
| 4 Nonfunctional (unintentionally present) | | | | ✓ |
| 5 | | | | |
| 6 | | | | |

| Replications | Definition [✓] | Data array [ ] | Includes | Excludes |
|---|---|---|---|---|
| 1 Master source statements (originals) | | | ✓ | |
| 2 Physical replicates of master statements, stored in the master code | | | ✓ | |
| 3 Copies inserted, instantiated, or expanded when compiling or linking | | | | ✓ |
| 4 Postproduction replicates—as in distributed, redundant, or reparameterized systems | | | | ✓ |
| 5 | | | | |

| Development status | Definition [✓] | Data array [ ] | Includes | Excludes |
|---|---|---|---|---|
| *Each statement has one and only one status, usually that of its parent unit.* | | | | |
| 1 Estimated or planned | | | | ✓ |
| 2 Designed | | | | ✓ |
| 3 Coded | | | | ✓ |
| 4 Unit tests completed | | | | ✓ |
| 5 Integrated into components | | | | ✓ |
| 6 Test readiness review completed | | | | ✓ |
| 7 Software (CSCI) tests completed | | | | ✓ |
| 8 System tests completed | | | ✓ | |
| 9 | | | | |
| 10 | | | | |
| 11 | | | | |

| Language | Definition [ ] | Data array [✓] | Includes | Excludes |
|---|---|---|---|---|
| *List each source language on a separate line.* | | | | |
| 1 | ***Separate totals for each language*** | | ✓ | |
| 2 Job control languages | | | | |
| 3 | | | | |
| 4 Assembly languages | | | | |
| 5 | | | | |
| 6 Third generation languages | | | | |
| 7 | | | | |
| 8 Fourth generation languages | | | | |
| 9 | | | | |
| 10 Microcode | | | | |
| 11 | | | | |

# Definition Checklist for Source Statement Counts

Definition name: ***Data Spec B+C: Project Analysis (combined specifications)***    Date: ***8/7/92***
Originator ***SEI***

**Measurement unit:** **Physical source lines** ☐ **Logical source statements** ☐

| **Statement type** **Definition** ☐ **Data array** ✓ | | **Includes** | **Excludes** |
|---|---|---|---|
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | |
| 1 Executable **Order of precedence ->** | 1 | ✓ | |
| 2 Nonexecutable | | | |
| 3 Declarations | 2 | ✓ | |
| 4 Compiler directives | 3 | ✓ | |
| 5 Comments | | | |
| 6 On their own lines | 4 | ✓ | |
| 7 On lines with source cod | 5 | ✓ | |
| 8 Banners and nonblank spacers | 6 | | ✓ |
| 9 Blank (empty) comments | 7 | | ✓ |
| 10 Blank lines | 8 | | ✓ |
| 11 | | | |
| 12 | | | |

| **How produced** **Definition** ☐ **Data array** ✓ | **Includes** | **Excludes** |
|---|---|---|
| 1 Programmed | ✓ | |
| 2 Generated with source code generators | ✓ | |
| 3 Converted with automated translators | ✓ | |
| 4 Copied or reused without change | ✓ | |
| 5 Modified | ✓ | |
| 6 Removed | ✓ | |
| 7 | | |
| 8 | | |

| **Origin** **Definition** ☐ **Data array** ✓ | **Includes** | **Excludes** |
|---|---|---|
| 1 New work: no prior existence | ✓ | |
| 2 Prior work: taken or adapted from | | |
| 3 A previous version, build, or release | ✓ | |
| 4 Commercial, off-the-shelf software (COTS), other than libraries | ✓ | |
| 5 Government furnished software (GFS), other than reuse libraries | ✓ | |
| 6 Another product | ✓ | |
| 7 A vendor-supplied language support library (unmodified) | | ✓ |
| 8 A vendor-supplied operating system or utility (unmodified) | | ✓ |
| 9 A local or modified language support library or operating system | ✓ | |
| 10 Other commercial library | ✓ | |
| 11 A reuse library (software designed for reus | ✓ | |
| 12 Other software component or library | ✓ | |
| 13 | | |
| 14 | | |

| **Usage** **Definition** ✓ **Data array** ☐ | **Includes** | **Excludes** |
|---|---|---|
| 1 In or as part of the primary product | ✓ | |
| 2 External to or in support of the primary product | | ✓ |
| 3 | | |

Definition name: ***Data Spec B+C: Project Analysis (combined specifications)***

| | | Includes | Excludes |
|---|---|---|---|
| **Delivery** | **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
| 1 | Delivered | | |
| 2 | Delivered as source | ✔ | |
| 3 | Delivered in compiled or executable form, but not as source | ✔ | |
| 4 | Not delivered | | |
| 5 | Under configuration control | | ✔ |
| 6 | Not under configuration control | | ✔ |
| 7 | | | |
| **Functionality** | **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
| 1 | Operative | ✔ | |
| 2 | Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | |
| 3 | Functional (intentional dead code, reactivated for special purposes) | ✔ | |
| 4 | Nonfunctional (unintentionally present) | | ✔ |
| 5 | | | |
| 6 | | | |
| **Replications** | **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
| 1 | Master source statements (originals) | ✔ | |
| 2 | Physical replicates of master statements, stored in the master code | ✔ | |
| 3 | Copies inserted, instantiated, or expanded when compiling or linking | | ✔ |
| 4 | Postproduction replicates—as in distributed, redundant, or reparameterized systems | | ✔ |
| 5 | | | |
| **Development status** | **Definition** [✓] **Data array** [ ] | **Includes** | **Excludes** |
| | *Each statement has one and only one status, usually that of its parent unit.* | | |
| 1 | Estimated or planned | | ✔ |
| 2 | Designed | | ✔ |
| 3 | Coded | | ✔ |
| 4 | Unit tests completed | | ✔ |
| 5 | Integrated into components | | ✔ |
| 6 | Test readiness review completed | | ✔ |
| 7 | Software (CSCI) tests completed | | ✔ |
| 8 | System tests completed | ✔ | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| **Language** | **Definition** [ ] **Data array** [✓] | **Includes** | **Excludes** |
| | *List each source language on a separate line.* | | |
| 1 | ***Separate totals for each language*** | ✔ | |
| 2 | Job control languages | | |
| 3 | | | |
| 4 | Assembly languages | | |
| 5 | | | |
| 6 | Third generation languages | | |
| 7 | | | |
| 8 | Fourth generation languages | | |
| 9 | | | |
| 10 | Microcode | | |
| 11 | | | |

## Rules for Counting Physical Source Lines

For each source language to which the definition applies, provide the following information:

**Language name**:

Note: This information is required only for statement types that are excluded from counts or for which individual counts are recorded.

| | |
|---|---|
| **Executable lines**: List the rules used to identify executable lines. If special rules are used for constructs such as block statements, embedded statements, empty statements, or embedded comments, describe them. | **Comments:** List the rules used to identify beginnings and endings of comments. |
| **Declarations**: List the rules used to identify declaration lines. Explain how declarations are distinguished from executable statements. | **Modified comments**: If separate counts are made for modified lines, list the rules used to keep modifications to comments on lines with other code from being classified as modified statements of higher precedence. |
| **Compiler directives**: List the rules used to identify compiler directives. | **Special rules**: List any special rules that are used to classify the first or last statements of any sections of code. |

## Rules for Counting Logical Source Statements

For each source language to which this definition applies, provide the following information:

**Language name**:

| | |
|---|---|
| **Executable statements**: List all rules and delimiters used to identify beginnings and endings of executable statements. If special rules are used for constructs such as block statements, embedded statements, empty statements, expression statements, or subprogram arguments, describe them. | **Comments:** If comments are counted, list the rules used to identify beginnings and endings of comment *statements*. Explain how, if at all, comment *statements* differ from physical source lines. |
| **Declarations**: List the rules and delimiters used to identify beginnings and endings of declarations. Explain how declarations are distinguished from executable statements. | **Special rules**: List any special rules or delimiters that are used to identify the first or last statements of any sections of code. |
| **Compiler directives**: List the rules and delimiters used to identify beginnings and endings of compiler directives. | **Exclusions:** List all keywords and symbols that, although set off by statement delimiters, are not counted as logical source statements. |

## Practices Used to Identify Inoperative Elements

List or explain the methods or rules used to identify:
**Intentionally bypassed statements and declarations**

**Unintentionally included dead code**

A. Unreachable, bypassed, or unreferenced elements (declarations, statements, or data stores) within modules:

B. Unused, unreferenced, or unaccessed modules or include files in code libraries:

C. Unused modules, procedures, or functions, linked into delivered products:

# Source Code Size Recording Form

❑ Physical source lines ❑ Logical source statements

Product name: ______________________ Version: ____________

Module name: ______________________ Version: ____________

Definition name: ***Data Spec A (project tracking)*** Dated: ***8/7/92***

Source language: ______________________ Max. line length: _____ characters

❑ Measured ❑ Estimated By: ______________________ Date: ____________

How measured (list tools used): ______________________

Inoperative code: ❑ Included ❑ Excluded, unless functional ❑ Excluded ❑ Don't know

If excluded, how identified: ______________________

Measurement results:

| | How produced | | | | | |
|---|---|---|---|---|---|---|
| Totals (excluding comments & blanks) | Programmed | Generated | Converted | Copied | Modified | Removed |
| | | | | | | |

| **Delivery** | | **Usage** | | **Development status** | |
|---|---|---|---|---|---|
| Delivered as source | ❑ | In primary product | ❑ | Estimated or planned | ❑ |
| Delivered as executable | ❑ | External to product | ❑ | Designed | ❑ |
| Not delivered, controlled | ❑ | | | Coded | ❑ |
| Not delivered, not controlled | ❑ | | | Unit tests completed | ❑ |
| | | | | Integrated into CSCs | ❑ |
| | | | | Test readiness reviewed | ❑ |
| | | | | CSCI tests completed | ❑ |
| **Module size** | | | | System tests completed | ❑ |

## Source Code Size Recording Form

☐ Physical source lines ☐ Logical source statements

| | | | |
|---|---|---|---|
| Product name: | | Version: | |
| Module name: | | Version: | |
| Definition name: | *Data Spec B (project analysis)* | Dated: | *8/7/92* |
| Source language: | | Max. line length: | characters |

☐ Measured ☐ Estimated By: ______ Date: ______

How measured (list tools used): ______

Inoperative code: ☐ Included ☐ Excluded, unless functional ☐ Excluded ☐ Don't know

If excluded, how identified: ______

Measurement results:

| | How produced | | | | | |
|---|---|---|---|---|---|---|
| **Statement type** | Programmed | Generated | Converted | Copied | Modified | Removed |
| Executable | | | | | | |
| Declaration | | | | | | |
| Compiler directive | | | | | | |
| Comment on own line | | | | | | |
| Comment on code line | | | | | | |

**Delivery**

| | |
|---|---|
| Delivered as source | ☐ |
| Delivered as executable | ☐ |
| Not delivered, controlled | ☐ |
| Not delivered, not controlled | ☐ |

**Usage**

| | |
|---|---|
| In primary product | ☐ |
| External to product | ☐ |

**Development status**

| | |
|---|---|
| Estimated or planned | ☐ |
| Designed | ☐ |
| Coded | ☐ |
| Unit tests completed | ☐ |
| Integrated into CSCs | ☐ |
| Test readiness reviewed | ☐ |
| CSCI tests completed | ☐ |
| System tests completed | ☐ |

**Module size** ______

## Source Code Size Recording Form

❑ Physical source lines ❑ Logical source statements

Product name: __________ Version: __________

Module name: __________ Version: __________

Definition name: *Data Spec C (reuse measurement)* Dated: *8/7/92*

Source language: __________ Max. line length: _____ characters

❑ Measured ❑ Estimated By: __________ Date: __________

How measured (list tools used): __________

Inoperative code: ❑ Included ❑ Excluded, unless functional ❑ Excluded ❑ Don't know

If excluded, how identified: __________

Measurement results (executable statements plus declarations plus compiler directives):

| | **How produced** | | | | | |
|---|---|---|---|---|---|---|
| **Origin** | Programmed | Generated | Converted | Copied | Modified | Removed |
| New: no prior existence | | | | | | |
| Previous version, build, or release | | | | | | |
| Commercial, off-the-shelf software | | | | | | |
| Government furnished software | | | | | | |
| Another product | | | | | | |
| Local or modified lang. library or O/S | | | | | | |
| Other commercial library | | | | | | |
| Reuse library | | | | | | |
| Other component or library | | | | | | |

| **Delivery** | | **Usage** | | **Development status** | |
|---|---|---|---|---|---|
| Delivered as source | ❑ | In primary product | ❑ | Estimated or planned | ❑ |
| Delivered as executable | ❑ | External to product | ❑ | Designed | ❑ |
| Not delivered, controlled | ❑ | | | Coded | ❑ |
| Not delivered, not controlled | ❑ | | | Unit tests completed | ❑ |
| | | | | Integrated into CSCs | ❑ |
| | | | | Test readiness reviewed | ❑ |
| | | | | CSCI tests completed | ❑ |
| | | | | System tests completed | ❑ |

**Module size** ☐ noncomment, nonblank statements

# Source Code Size Recording Form

❑ Physical source lines ❑ Logical source statements

Product name: ______________________ Version: ____________

Module name: ______________________ Version: ____________

Definition name: ______________________ Dated: ____________

Source language: ______________________ Max. line length: _____ characters

❑ Measured ❑ Estimated By: ______________________ Date: ____________

How measured (list tools used): ______________________

Inoperative code: ❑ Included ❑ Excluded, unless functional ❑ Excluded ❑ Don't know

If excluded, how identified: ______________________

Measurement results:

| | How produced | | | | | |
|---|---|---|---|---|---|---|
| **Statement type** | Programmed | Generated | Converted | Copied | Modified | Removed |
| Executable | | | | | | |
| Declarations | | | | | | |
| Compiler directives | | | | | | |
| Comments | | | | | | |
| on their own lines | | | | | | |
| on lines with code | | | | | | |
| banners & spacers | | | | | | |
| blank comments | | | | | | |
| Blank lines | | | | | | |
| | | | | | | |

| **Origin** | | **Delivery** | | **Development status** | |
|---|---|---|---|---|---|
| New: no prior existence | | Delivered as source | ❑ | Estimated or planned | ❑ |
| Previous version, build, or release | | Delivered as executable | ❑ | Designed | ❑ |
| Commercial, off-the-shelf software | | Not delivered, controlled | ❑ | Coded | ❑ |
| Government furnished software | | Not delivered, | | Unit tests completed | ❑ |
| Another product | | not controlled | ❑ | Integrated into CSCs | ❑ |
| Vendor-supplied language library | | | | Test readiness reviewed | ❑ |
| Vendor-supplied O/S (unmodified) | | **Usage** | | CSCI tests completed | ❑ |
| Local or modified lang. library or O/S | | In primary product | ❑ | System tests completed | ❑ |
| Other commercial library | | External to product | ❑ | | |
| Reuse library | | | | | |
| Other component or library | | | | **Module size** | |

# Data Summary—Source Code Size

Module ID ______________________ Date counted: ____________

Language: ______________________ Reported by: ____________

| **Measured as:** ☐ **Physical source lines** ☐ **Logical statements** | **Counted** ____ **Estimated** ____ **Total** ____ | | **Totals include** | **Totals exclude** | **Individual totals** |
|---|---|---|---|---|---|
| **Statement type** | | | | | |
| *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | | | |
| 1 Executable | **Order of precedence ->** | 1 | | | |
| 2 Nonexecutable | | | | | |
| 3 Declarations | | 2 | | | |
| 4 Compiler directives | | 3 | | | |
| 5 Comments | | | | | |
| 6 On their own lines | | 4 | | | |
| 7 On lines with source code | | 5 | | | |
| 8 Banners and nonblank spacers | | 6 | | | |
| 9 Blank (empty) comments | | 7 | | | |
| 10 Blank lines | | 8 | | | |
| 11 | | | | | |
| 12 | | | | | |
| **How produced** | | | | | |
| 1 Programmed | | | | | |
| 2 Generated with source code generators | | | | | |
| 3 Converted with automated translators | | | | | |
| 4 Copied or reused without change | | | | | |
| 5 Modified | | | | | |
| 6 Removed | | | | | |
| 7 | | | | | |
| 8 | | | | | |
| **Origin** | | | | | |
| 1 New work: no prior existence | | | | | |
| 2 Prior work: taken or adapted from | | | | | |
| 3 A previous version, build, or release | | | | | |
| 4 Commercial, off-the-shelf software (COTS), other than libraries | | | | | |
| 5 Government furnished software (GFS), other than reuse librar | | | | | |
| 6 Another product | | | | | |
| 7 A vendor-supplied language support library (unmodified) | | | | | |
| 8 A vendor-supplied operating system or utility (unmodified) | | | | | |
| 9 A local or modified language support library or operating system | | | | | |
| 10 Other commercial library | | | | | |
| 11 A reuse library (software designed for reuse) | | | | | |
| 12 Other software component or library | | | | | |
| 13 | | | | | |
| 14 | | | | | |
| **Usage** | | | | | |
| 1 In or as part of the primary product | | | | | |
| 2 External to or in support of the primary produ | | | | | |
| 3 | | | | | |

| Module ID ________ Language: ________ | Totals include | Totals exclude | Individual totals |
|---|---|---|---|
| **Delivery** | | | |
| 1 Delivered | | | |
| 2 Delivered as source | | | |
| 3 Delivered in compiled or executable form, but not as source | | | |
| 4 Not delivered | | | |
| 5 Under configuration control | | | |
| 6 Not under configuration control | | | |
| 7 | | | |
| **Functionality** | | | |
| 1 Operative | | | |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | | |
| 3 Functional (intentional dead code, reactivated for special purposes) | | | |
| 4 Nonfunctional (unintentionally present) | | | |
| 5 | | | |
| 6 | | | |
| **Replications** | | | |
| 1 Master source statements (originals) | | | |
| 2 Physical replicates of master statements, stored in the master code | | | |
| 3 Copies inserted, instantiated, or expanded when compiling or linking | | | |
| 4 Postproduction replicates—as in distributed, redundant, | | | |
| or reparameterized systems | | | |
| 5 | | | |
| **Development status** | | | |
| *Each statement has one and only one status, usually that of its parent unit.* | | | |
| 1 Estimated or planned | | | |
| 2 Designed | | | |
| 3 Coded | | | |
| 4 Unit tests completed | | | |
| 5 Integrated into components | | | |
| 6 Test readiness review completed | | | |
| 7 Software (CSCI) tests completed | | | |
| 8 System tests completed | | | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| **Language** | | | |
| *List each source language on a separate line.* | | | |
| 1 ________ | | | |
| 2 Job control languages ________ | | | |
| 3 ________ | | | |
| 4 Assembly languages ________ | | | |
| 5 ________ | | | |
| 6 Third generation languages ________ | | | |
| 7 ________ | | | |
| 8 Fourth generation languages ________ | | | |
| 9 ________ | | | |
| 10 Microcode ________ | | | |
| 11 | | | |

| Module ID ______ Language: ______ | | Totals include | Totals exclude | Individual totals |
|---|---|---|---|---|
| **Clarifications (general)** **Listed elements are assigned to statement type –>** | | | | |
| 1 Nulls, continues, and no-ops | | | | |
| 2 Empty statements (e.g., “;;” and lone semicolons on separate lines) | | | | |
| 3 Statements that instantiate generics | | | | |
| 4 Begin…end and {…} pairs used as executable statements | | | | |
| 5 Begin…end and {…} pairs that delimit (sub)program bodies | | | | |
| 6 Logical expressions used as test conditions | | | | |
| 7 Expression evaluations used as subprogram arguments | | | | |
| 8 End symbols that terminate executable statements | | | | |
| 9 End symbols that terminate declarations or (sub)program boc | | | | |
| 10 Then, else, and otherwise symbols | | | | |
| 11 Elseif statements | | | | |
| 12 Keywords like procedure division, interface, and implementation | | | | |
| 13 Labels (branching destinations) on lines by themselves | | | | |
| 14 | | | | |
| 15 | | | | |
| 16 | | | | |
| **Clarifications (language specific)** | | | | |
| **Ada** | | | | |
| 1 End symbols that terminate declarations or (sub)program boc | | | | |
| 2 Block statements (e.g., begin…end) | | | | |
| 3 With and use clauses | | | | |
| 4 When (the keyword preceding executable statements) | | | | |
| 5 Exception (the keyword, used as a frame header) | | | | |
| 6 Pragmas | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| **Assembly** | | | | |
| 1 Macro calls | | | | |
| 2 Macro expansions | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| **C and C++** | | | | |
| 1 Null statement (e.g., “;” by itself to indicate an empty body) | | | | |
| 2 Expression statements (expressions terminated by semicolon | | | | |
| 3 Expressions separated by semicolons, as in a "for" statement | | | | |
| 4 Block statements (e.g., {…} with no terminating semicolon) | | | | |
| 5 “{”, “}”, or “};” on a line by itself when part of a declaration | | | | |
| 6 “{” or “}” on line by itself when part of an executable statement | | | | |
| 7 Conditionally compiled statements (#if, #ifdef, #ifndef) | | | | |
| 8 Preprocessor statements other than #if, #ifdef, and #ifndef | | | | |
| 9 | | | | |
| 10 | | | | |
| 11 | | | | |
| 12 | | | | |

| Module ID ______ Language: ______ | | **Totals include** | **Totals exclude** | **Individual totals** |
|---|---|---|---|---|
| **CMS-2** | **Listed elements are assigned to** | | | |
| 1 Keywords like SYS-PROC and SYS-DD | **statement type –>** | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| **COBOL** | | | | |
| 1 “PROCEDURE DIVISION”, “END DECLARATIVES”, etc. | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| **FORTRAN** | | | | |
| 1 END statements | | | | |
| 2 Format statements | | | | |
| 3 Entry statements | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| **JOVIAL** | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| **Pascal** | | | | |
| 1 Executable statements not terminated by semicolons | | | | |
| 2 Keywords like INTERFACE and IMPLEMENTATION | | | | |
| 3 FORWARD declarations | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |

| Module ID ________<br>Language: ________ | | Totals include | Totals exclude | Individual totals |
|---|---|---|---|---|
| **Listed elements are assigned to statement type –>** | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| 10 | | | | |
| 11 | | | | |
| 12 | | | | |

**Summary of Statement Types**

**Executable statements**

Executable statements cause runtime actions. They may be simple statements such as assignments, goto's, procedure calls, macro calls, returns, breaks, exits, stops, continues, nulls, no-ops, empty statements, and FORTRAN's END. Or they may be structured or compound statements, such as conditional statements, repetitive statements, and "with" statements. Languages like Ada, C, C++, and Pascal have block statements [begin…end and {…}] that are classified as executable when used where other executable statements would be permitted. C and C++ define expressions as executable statements when they terminate with a semicolon, and C++ has a <declaration> statement that is executable.

**Declarations**

Declarations are nonexecutable program elements that affect an assembler's or compiler's interpretation of other program elements. They are used to name, define, and initialize; to specify internal and external interfaces; to assign ranges for bounds checking; and to identify and bound modules and sections of code. Examples include declarations of names, numbers, constants, objects, types, subtypes, programs, subprograms, tasks, exceptions, packages, generics, macros, and deferred constants. Declarations also include renaming declarations, use clauses, and declarations that instantiate generics. Mandatory begin…end and {…} symbols that delimit bodies of programs and subprograms are integral parts of program and subprogram declarations. Language superstructure elements that establish boundaries for different sections of source code are also declarations. Examples include terms such as PROCEDURE DIVISION, DATA DIVISION, DECLARATIVES, END DECLARATIVES, INTERFACE, IMPLEMENTATION, SYS-PROC, and SYS-DD. Declarations, in general, are never required by language specifications to initiate runtime actions, although some languages permit compilers to implement them that way.

**Compiler Directives**

Compiler directives instruct compilers, preprocessors, or translators (but not runtime systems) to perform special actions. Some, such as Ada's pragma and COBOL's COPY, REPLACE, and USE, are integral parts of the source language. In other languages like C and C++, special symbols like # are used along with standardized keywords to direct preprocessor or compiler actions. Still other languages rely on nonstandardized methods supplied by compiler vendors. In these languages, directives are often designated by special symbols such as #, $, an

# Source Code Size Reporting Form

❑ Physical source lines　　❑ Logical source statements

Product or module name: ______________________ Version: __________

Definition name: *Data Spec A (project tracking)* Dated: *7/9/92*

Source language: ______________________ Max. line length: _____ characters

❑ Measured　❑ Estimated　By: ______________________ Date: __________

How measured (list tools used): ______________________

Inoperative code: ❑ Included　❑ Excluded, unless functional　❑ Excluded　❑ Don't know

If excluded, how identified: ______________________

| **Delivery** | | **Usage** | | **Total Removed** |
|---|---|---|---|---|
| Delivered | ❑ | In primary product | ❑ | |
| Not delivered | ❑ | External to product | ❑ | |

Measurement results:

| | **How produced** | | | | | |
|---|---|---|---|---|---|---|
| **Development Status** | Programmed | Generated | Converted | Copied | Modified | Total |
| Coded | | | | | | |
| Unit tests completed | | | | | | |
| Integrated into CSCs | | | | | | |
| Test readiness reviewed | | | | | | |
| CSCI tests completed | | | | | | |
| System tests completed | | | | | | |
| Total | | | | | | |

# Data Summary Request—Source Code Size

Definition name: ______________________ Date: ____________

______________________ Originator: ____________

| **Measured as:** | **Counted** / **Estimated** / **Total** | | **Totals include** | **Totals exclude** | **Individual totals** |
|---|---|---|---|---|---|
| ☐ **Physical source lines** | **Counted** | | | | |
| ☐ **Logical statements** | **Estimated** | | | | |
| | **Total** | | | | |

| | | Order of precedence | Totals include | Totals exclude | Individual totals |
|---|---|---|---|---|---|
| **Statement type** *When a line or statement contains more than one type, classify it as the type with the highest precedence.* | | | | | |
| 1 | Executable **Order of precedence ->** | 1 | | | |
| 2 | Nonexecutable | | | | |
| 3 | Declarations | 2 | | | |
| 4 | Compiler directives | 3 | | | |
| 5 | Comments | | | | |
| 6 | On their own lines | 4 | | | |
| 7 | On lines with source code | 5 | | | |
| 8 | Banners and nonblank spacers | 6 | | | |
| 9 | Blank (empty) comments | 7 | | | |
| 10 | Blank lines | 8 | | | |
| 11 | | | | | |
| 12 | | | | | |
| **How produced** | | | | | |
| 1 | Programmed | | | | |
| 2 | Generated with source code generators | | | | |
| 3 | Converted with automated translators | | | | |
| 4 | Copied or reused without change | | | | |
| 5 | Modified | | | | |
| 6 | Removed | | | | |
| 7 | | | | | |
| 8 | | | | | |
| **Origin** | | | | | |
| 1 | New work: no prior existence | | | | |
| 2 | Prior work: taken or adapted from | | | | |
| 3 | A previous version, build, or release | | | | |
| 4 | Commercial, off-the-shelf software (COTS), other than libraries | | | | |
| 5 | Government furnished software (GFS), other than reuse librar | | | | |
| 6 | Another product | | | | |
| 7 | A vendor-supplied language support library (unmodified) | | | | |
| 8 | A vendor-supplied operating system or utility (unmodified) | | | | |
| 9 | A local or modified language support library or operating system | | | | |
| 10 | Other commercial library | | | | |
| 11 | A reuse library (software designed for reuse) | | | | |
| 12 | Other software component or library | | | | |
| 13 | | | | | |
| 14 | | | | | |
| **Usage** | | | | | |
| 1 | In or as part of the primary product | | | | |
| 2 | External to or in support of the primary produ | | | | |
| 3 | | | | | |

| Definition name: ______________________ ______________________ | Totals include | Totals exclude | Individual totals |
|---|---|---|---|
| **Delivery** | | | |
| 1 Delivered | | | |
| 2 Delivered as source | | | |
| 3 Delivered in compiled or executable form, but not as source | | | |
| 4 Not delivered | | | |
| 5 Under configuration control | | | |
| 6 Not under configuration control | | | |
| 7 | | | |
| **Functionality** | | | |
| 1 Operative | | | |
| 2 Inoperative (dead, bypassed, unused, unreferenced, or unaccessed) | | | |
| 3 Functional (intentional dead code, reactivated for special purposes) | | | |
| 4 Nonfunctional (unintentionally present) | | | |
| 5 | | | |
| 6 | | | |
| **Replications** | | | |
| 1 Master source statements (originals) | | | |
| 2 Physical replicates of master statements, stored in the master code | | | |
| 3 Copies inserted, instantiated, or expanded when compiling or linking | | | |
| 4 Postproduction replicates—as in distributed, redundant, or reparameterized systems | | | |
| 5 | | | |
| **Development status** *Each statement has one and only one status, usually that of its parent unit.* | | | |
| 1 Estimated or planned | | | |
| 2 Designed | | | |
| 3 Coded | | | |
| 4 Unit tests completed | | | |
| 5 Integrated into components | | | |
| 6 Test readiness review completed | | | |
| 7 Software (CSCI) tests completed | | | |
| 8 System tests completed | | | |
| 9 | | | |
| 10 | | | |
| 11 | | | |
| **Language** *List each source language on a separate line.* | | | |
| 1 | | | |
| 2 Job control languages ______ | | | |
| 3 ______ | | | |
| 4 Assembly languages ______ | | | |
| 5 ______ | | | |
| 6 Third generation languages ______ | | | |
| 7 ______ | | | |
| 8 Fourth generation languages ______ | | | |
| 9 ______ | | | |
| 10 Microcode ______ | | | |
| 11 | | | |

| Definition name: ______________________ ______________________ | | Totals include | Totals exclude | Individual totals |
|---|---|---|---|---|
| **Clarifications (general)** **Listed elements are assigned to statement type –>** | | | | |
| 1 Nulls, continues, and no-ops | | | | |
| 2 Empty statements (e.g., “;;” and lone semicolons on separate lines) | | | | |
| 3 Statements that instantiate generics | | | | |
| 4 Begin…end and {…} pairs used as executable statements | | | | |
| 5 Begin…end and {…} pairs that delimit (sub)program bodies | | | | |
| 6 Logical expressions used as test conditions | | | | |
| 7 Expression evaluations used as subprogram arguments | | | | |
| 8 End symbols that terminate executable statements | | | | |
| 9 End symbols that terminate declarations or (sub)program boc | | | | |
| 10 Then, else, and otherwise symbols | | | | |
| 11 Elseif statement: | | | | |
| 12 Keywords like procedure division, interface, and implementation | | | | |
| 13 Labels (branching destinations) on lines by themselves | | | | |
| 14 | | | | |
| 15 | | | | |
| 16 | | | | |
| **Clarifications (language specific)** | | | | |
| **Ada** | | | | |
| 1 End symbols that terminate declarations or (sub)program boc | | | | |
| 2 Block statements (e.g., begin…end) | | | | |
| 3 With and use clauses | | | | |
| 4 When (the keyword preceding executable statements) | | | | |
| 5 Exception (the keyword, used as a frame header) | | | | |
| 6 Pragmas | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| **Assembly** | | | | |
| 1 Macro calls | | | | |
| 2 Macro expansions | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| **C and C++** | | | | |
| 1 Null statement (e.g., “;” by itself to indicate an empty body) | | | | |
| 2 Expression statements (expressions terminated by semicolon | | | | |
| 3 Expressions separated by semicolons, as in a "for" statement | | | | |
| 4 Block statements (e.g., {…} with no terminating semicolon) | | | | |
| 5 “{”, “}”, or “};” on a line by itself when part of a declaration | | | | |
| 6 “{” or “}” on line by itself when part of an executable statement | | | | |
| 7 Conditionally compiled statements (#if, #ifdef, #ifndef) | | | | |
| 8 Preprocessor statements other than #if, #ifdef, and #ifndef | | | | |
| 9 | | | | |
| 10 | | | | |
| 11 | | | | |
| 12 | | | | |

| Definition name: ______________________ | | Totals include | Totals exclude | Individual totals |
|---|---|---|---|---|
| **CMS-2** **Listed elements are assigned to** | | | | |
| 1 Keywords like SYS-PROC and SYS-DD **statement type –>** | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| **COBOL** | | | | |
| 1 "PROCEDURE DIVISION", "END DECLARATIVES", etc. | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| **FORTRAN** | | | | |
| 1 END statements | | | | |
| 2 Format statements | | | | |
| 3 Entry statements | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| **JOVIAL** | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| **Pascal** | | | | |
| 1 Executable statements not terminated by semicolons | | | | |
| 2 Keywords like INTERFACE and IMPLEMENTATION | | | | |
| 3 FORWARD declarations | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |

| Definition name: | | Totals include | Totals exclude | Individual totals |
|---|---|---|---|---|
| Listed elements are assigned to statement type –> | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| 10 | | | | |
| 11 | | | | |
| 12 | | | | |

**Summary of Statement Types**

**Executable statements**

Executable statements cause runtime actions. They may be simple statements such as assignments, goto's, procedure calls, macro calls, returns, breaks, exits, stops, continues, nulls, no-ops, empty statements, and FORTRAN's END. Or they may be structured or compound statements, such as conditional statements, repetitive statements, and "with" statements. Languages like Ada, C, C++, and Pascal have block statements [begin…end and {…}] that are classified as executable when used where other executable statements would be permitted. C and C++ define expressions as executable statements when they terminate with a semicolon, and C++ has a <declaration> statement that is executable.

**Declarations**

Declarations are nonexecutable program elements that affect an assembler's or compiler's interpretation of other program elements. They are used to name, define, and initialize; to specify internal and external interfaces; to assign ranges for bounds checking; and to identify and bound modules and sections of code. Examples include declarations of names, numbers, constants, objects, types, subtypes, programs, subprograms, tasks, exceptions, packages, generics, macros, and deferred constants. Declarations also include renaming declarations, use clauses, and declarations that instantiate generics. Mandatory begin…end and {…} symbols that delimit bodies of programs and subprograms are integral parts of program and subprogram declarations. Language superstructure elements that establish boundaries for different sections of source code are also declarations. Examples include terms such as PROCEDURE DIVISION, DATA DIVISION, DECLARATIVES, END DECLARATIVES, INTERFACE, IMPLEMENTATION, SYS-PROC, and SYS-DD. Declarations, in general, are never required by language specifications to initiate runtime actions, although some languages permit compilers to implement them that way.

**Compiler Directives**

Compiler directives instruct compilers, preprocessors, or translators (but not runtime systems) to perform special actions. Some, such as Ada's pragma and COBOL's COPY, REPLACE, and USE, are integral parts of the source language. In other languages like C and C++, special symbols like # are used along with standardized keywords to direct preprocessor or compiler actions. Still other languages rely on nonstandardized methods supplied by compiler vendors. In these languages, directives are often designated by special symbols such as #, $, an

# REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a. REPORT SECURITY CLASSIFICATION<br>Unclassified | 1b. RESTRICTIVE MARKINGS<br>None |
| 2a. SECURITY CLASSIFICATION AUTHORITY<br>N/A | 3. DISTRIBUTION/AVAILABILITY OF REPORT<br>Approved for Public Release<br>Distribution Unlimited |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE<br>N/A | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S)<br>CMU/SEI-92-TR-20 | 5. MONITORING ORGANIZATION REPORT NUMBER(S)<br>ESC-TR-92-020 |

| | | |
|---|---|---|
| 6a. NAME OF PERFORMING ORGANIZATION<br>Software Engineering Institute | 6b. OFFICE SYMBOL<br>(if applicable)<br>SEI | 7a. NAME OF MONITORING ORGANIZATION<br>ESC/AVS<br>Hanscom Air Force Base, MA 01731 |
| 6c. ADDRESS (City, State and ZIP Code)<br>Carnegie Mellon University<br>Pittsburgh, PA 15213 | | 7b. ADDRESS (City, State and ZIP Code)<br>ESC/AVS<br>Hanscom Air Force Base, MA 01731 |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION<br>SEI Joint Program Office | 8B. OFFICE SYMBOL<br>( if applicable)<br>ESD/AVS | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER<br>F1962890C0003 |

8c. ADDRESS (City, State and ZIP Code)
Carnegie Mellon University
Pittsburgh, PA 15213

10. SOURCE OF FUNDING NOS.

| PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT NO. |
|---|---|---|---|
| 63756E | N/A | N/A | N/A |

11. TITLE (Include Security Classification)
Software Size Measurement:  A Framework for Counting Source Statements

12. PERSONAL AUTHOR (S)
Robert E. Park, et al

| 13a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT Yr., Mo., Day) | 15. PAGE COUNT |
|---|---|---|---|
| Final | FROM TO | September 1992 | 210 |

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

| FIELD | GROUP | SUB. GR. |
|---|---|---|
| | | |
| | | |
| | | |

18. SUBJECT TERMS (Continue on reverse iif necessary and identify by block number)
software metrics, software size, software measure, software measurement, lines of code, LOC, source lines of code, SLOC, source statements, source code size, software planning and tracking

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report presents guidelines for defining, recording, and reporting two frequently used measures of software size-physical source lines and logical source statements.  We propose a general framework for constructing size definitions and use it to derive operational methods can be applied to address the information needs of different users while maintaining a common definition of software size.

| | |
|---|---|
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT<br>Unclassified/unlimited same as RPTDTIC users | 21. ABSTRACT SECURITY CLASSIFICATION<br>Unclassified, Unlimited Distribution |

| 22a. NAME OF RESPONSIBLE INDIVIDUAL | 22b. TELEPHONE NUMBER (Include Area Code) | 22c. OFFICE SYMBOL |
|---|---|---|
| | | ESC/AVS (SEI) |