Software Development Cost Estimating Guidebook

Software Technology Support Center Cost Analysis Group

July 2009

Resource manual for education and support in developing credible software development cost estimates

Executive Summary

The purpose of the Software Development Estimating Guidebook is to provide the cost analyst with a resource manual to use in developing credible software development cost estimates. A realistic estimate is based upon a solid understanding of the software development process and the historical data that forms a framework for the expected values. An estimating methodology that follows a proven process consistent with best practices and Department of Defense (DoD) policies further contributes to estimate validity.

The information is presented at two levels. One level will help the experienced analyst immediately focus on the material necessary to develop an estimate. The second level of information is for the novice, or infrequent user, to use as educational information regarding the software development and estimating processes.

The estimating process starts with a determination of the purpose of the estimate. Next, the cost (or effort) and schedule for the software development project are determined using three factors: effective size, development environment, and product complexity.

The key, and most important, element in the software estimate is the effective size of the software product. Determining size can be approached from several directions depending upon the software size measure (lines of code, function points, use cases, etc.) used by the development organization. A system developed by writing lines of code requires a different estimating approach than a previously developed or off-the-shelf application. The acquisition phase also influences the analyst's approach because of the amount and type of software development data available from the program or developers.

The development environment is the next most important effort and schedule driver. The environment can be factored into five categories: (1) developer capability or efficiency, (2) personnel experience, (3) development system characteristics, (4) management characteristics, and (5) product characteristics. The last four categories are largely driven by the product requirements. These factors take into consideration the development environment itself, the capabilities and experience of the developers, the developing organization's management style, security requirements, and so on. These factors, along with software size and complexity, combine to determine the productivity or efficiency with which a developer can "build" and test the software. Ultimately, these environment characteristics drive the cost and schedule of the software development and implementation of the system.

It is uncertain who first coined the phrase, "A fool with a tool is still a fool." Plugging numbers into a parametric model without knowing if the results are realistic fits this adage. This guidebook addresses estimate realism using historical data, industry best practices, and authoritative insight. The insight comes from experts in the fields of software development and cost estimating. This information helps the analyst conduct a "sanity check" of their estimate results. A well-understood and validated estimate offers a defensible position for program office analysts, component cost agency analysts, and independent evaluators. A reasonable estimate is useful in budgeting, milestone decision reviews, and determining the life cycle or other costs of the program.

The contents of this guide, ten sections and nine appendices, are grouped into four major parts. An introduction and the basics of the software development process lead off the tutorial. The next two major parts cover the estimating process and related details. Finally, concepts and examples presented in the sections are expanded in a set of appendices. The idea behind this structure is to present principles for instruction and reference in the core sections and, then, examine details and related examples.

The information herein is not intended to dictate policy or supplant guidance given in official documents. However, the authors hope that everyone within the software cost estimating community in both the public and private sectors will find it useful. The extent of information on software development and cost estimating presented within these pages is not intended to be all-inclusive. Yet, the guidebook is meant to be comprehensive and complete, providing a single-resource document for use in creating estimates.

Table of Contents

Executive Summary ..... **i**

List of Figures ..... **xi**

List of Tables ..... **xii**

List of Equations ..... **xv**

Acknowledgements ..... **xvii**

Section 1 Introduction **1**

1.1 Development constraints 4

1.2 Major cost factors 4

1.2.1 Effective size 5

1.2.2 Product complexity 5

1.2.3 Development environment 6

1.2.4 Product characteristics 6

1.3 Software support estimation 6

1.4 Guidebook overview 7

Section 2 Software Development Process **10**

2.1 The Defense Acquisition System 10

2.1.1 Framework elements 10

2.1.2 User needs and technology opportunities 11

2.1.3 Pre-systems acquisition 11

2.1.3.1 Concept refinement phase 11

2.1.3.2 Milestone A 12

2.1.3.3 Technology development phase 12

2.1.3.4 Milestone B 12

2.1.4 Systems acquisition 12

2.1.4.1 System development & demonstration 12

2.1.4.2 Milestone C 12

2.1.4.3 Production & deployment 13

2.1.5 Sustainment 13
2.2 Waterfall model 13
2.2.1 Requirements analysis and specification 15
2.2.2 Full-Scale Development 16
2.2.3 System Integration and Test 17
2.3 Software development products 17
2.3.1 Software Development Plan 17
2.3.1.1 Project organization 17
2.3.1.2 Schedule 18
2.3.1.3 Software Design Document 18
2.3.1.4 Quality Plan 18
2.3.2 Software Requirements Specification 19
2.3.3 Interface Control Document 19

Section 3 Levels of Detail in Software Estimates 20
3.1 Estimate foundation factors 20
3.2 System-level estimating model 21
3.3 Component-level estimating model 23
3.4 Estimating process 26

Section 4 System-Level Estimating Process 27
4.1 Product complexity 28
4.2 Size estimating process 29
4.2.1 Effective source lines of code (ESLOC) 29
4.2.2 Function point counting 30
4.3 Software size growth 30
4.4 Productivity factor 31
4.4.1 Productivity factor table 32
4.4.2 ESC metrics 32
4.5 System-level cost estimating 33
4.6 Reality check 34
4.7 Allocate development effort 34
4.8 Allocate maintenance effort 36

4.8.1 Software enhancement 36
4.8.2 Knowledge retention 37
4.8.3 Steady state maintenance effort 38

Section 5 Component-Level Estimating Process 39
5.1 Staffing profiles 41
5.2 Product complexity 42
5.3 Size estimating process 42
5.4 Development environment 43
5.4.1 Personnel evaluation 43
5.4.2 Development environment evaluation 43
5.4.3 Product impact evaluation 43
5.4.4 Basic technology constant 44
5.4.5 Effective technology constant 44
5.5 Development cost and schedule calculations 45
5.6 Verify estimate realism 46
5.7 Allocate development effort and schedule 46
5.7.1 Effort allocation 46
5.7.2 Schedule allocation 48
5.8 Allocate maintenance effort 48

Section 6 Estimating Effective Size 49
6.1 Source code elements 50
6.1.1 Black box vs. white box elements 50
6.1.2 NEW source code 51
6.1.3 MODIFIED source code 51
6.1.4 DELETED source code 51
6.1.5 REUSED source code 51
6.1.6 COTS software 51
6.1.7 Total SLOC 52
6.2 Size uncertainty 52
6.3 Source line of code (SLOC) 53
6.3.1 Executable 53
6.3.2 Data declaration 54

6.3.3 Compiler directives 54
6.3.4 Format statements 54
6.4 Effective source lines of code (ESLOC) 54
6.4.1 Effective size as work 54
6.4.2 Effective size equation 56
6.4.2.1 Design factor 57
6.4.2.2 Implementation factor 57
6.4.2.3 Test factor 57
6.5 Size growth 58
6.6 Size risk 62
6.7 Function points 63
6.7.1 Function point counting 63
6.7.2 Function point components 64
6.7.2.1 Application boundary 65
6.7.2.2 Internal Logical File 66
6.7.2.3 External Interface File 19
6.7.2.4 External Input 68
6.7.2.5 External Output 68
6.7.2.6 External Inquiry 69
6.7.2.7 Transforms 69
6.7.2.8 Transitions 70
6.7.3 Unadjusted function point counting 71
6.7.4 Adjusted function points 71
6.7.4.1 Value Adjustment Factor 72
6.7.4.2 Adjusted function point calculation 73
6.7.5 Backfiring 73
6.7.6 Function points and objects 74
6.7.7 Zero Function point problem 75

Section 7 Productivity Factor Evaluation 77
7.1 Introduction 77
7.2 Determining productivity factor 78
7.2.1 ESC metrics 80
7.2.2 Productivity index 82

7.3 System-level estimating 83

Section 8 Evaluating Developer Capability 85

8.1 Importance of developer capability 85

8.2 Basic technology constant 86

8.2.1 Basic technology constant parameters 87

8.2.1.1 Analyst capability 87

8.2.1.2 Programmer capability 89

8.2.1.3 Application domain experience 89

8.2.1.4 Learning curve 91

8.2.1.5 Domain experience rating 91

8.2.1.6 Modern practices 91

8.2.1.7 Modern tools 92

8.2.2 Basic technology constant calculation 93

8.3 Mechanics of communication 94

8.3.1 Information convection 95

8.3.2 Radiation 96

8.3.3 Communication barriers 96

8.3.3.1 Skunk Works 96

8.3.3.2 Cube farm 97

8.3.3.3 Project area 97

8.3.4 Utensils for creative work 97

Section 9 Development Environment Evaluation 99

9.1 Learning curve vs. volatility 100

9.2 Personnel experience characteristics 100

9.2.1 Programming language experience 101

9.2.2 Practices and methods experience 103

9.2.3 Development system experience 103

9.2.4 Target system experience 104

9.3 Development support characteristics 104

9.3.1 Development system volatility 104

9.3.2 Practices/methods volatility 105

9.4 Management characteristics 105

9.4.1 Multiple security classifications 105
9.4.2 Multiple development organizations 105
9.4.3 Multiple development sites 106
9.4.4 Resources and support location 106

Section 10 Product Characteristics Evaluation 108
10.1 Product complexity 108
10.2 Display requirements 110
10.3 Rehosting requirements 111
10.4 Memory constraints 111
10.5 Required reliability 112
10.6 Real-time performance requirements 113
10.7 Requirements volatility 113
10.8 Security requirements 115

Appendix A Acronyms 116

Appendix B Terminology 122

Appendix C Bibliography 129

Appendix D Software Life Cycle Approaches 136
D.1 Waterfall 136
D.2 Spiral development 137
D.3 Evolutionary development 137
D.4 Incremental development 138
D.5 Agile development (Extreme programming) 138
D.6 Rapid application development 139
D.7 Other approaches 139

Appendix E Software Estimating Models 141
E.1 Analogy models 142
E.2 Expert judgment models 142
E.2.1 Delphi method 143
E.2.2 Wideband delphi method 144
E.3 Bottom-up estimating 144

E.4 Parametric models 145
E.5 Origins and evolution of parametric software models 145
E.6 First-order models 146
E.7 Second-order models 148
E.8 Third-order models 149

Appendix F System-Level Estimate Case Study 152
F.1 HACS baseline size estimate 152
F.2 HACS size growth calculation 155
F.3 HACS effort calculation 156
F.4 HACS Reality check 158
F.5 HACS development effort allocation 159
F.6 HACS maintenance effort calculation 160

Appendix G Component-Level Estimate Case Study 162
G.1 HACS baseline size estimate 162
G.2 HACS size estimate 163
G.3 HACS size growth calculation 165
G.4 HACS environment 167
G.4.1 HACS developer capability 167
G.4.2 Personnel evaluation 167
G.4.3 Development environment 168
G.4.4 HACS product impact 168
G.4.5 HACS effective technology constant 169
G.5 HACS development effort and schedule calculations 169
G.6 Verify HACS estimate realism 171
G.7 Allocate HACS development effort and schedule 172
G.7.1 HACS effort allocation 172
G.7.2 Schedule allocation 174
G.8 Allocate HACS maintenance effort 175

Appendix H The Defense Acquisition System 178
H.1 Basic definitions 179
H.2 Acquisition authorities 179
H.3 Acquisition categories 179

H.3.1 ACAT I 180
H.3.2 ACAT II 180
H.3.3 ACAT III 181
H.3.4 ACAT IV 181
H.3.5 Abbreviated Acquisition Programs (AAPs) 181
H.4 Acquisition management framework 181
H.4.1 Framework elements 182
H.4.2 User needs and technology opportunities 182
H.4.3 Pre-systems acquisition 183
H.4.3.1 Concept refinement phase 183
H.4.3.2 Milestone A 184
H.4.3.3 Technology development phase 184
H.4.3.4 Milestone B 185
H.4.4 Systems acquisition 185
H.4.4.1 System development and demonstration 185
H.4.4.2 Milestone C 186
H.4.4.3 Production and deployment 186
H.4.5 Sustainment 187
H.4.5.1 Sustainment effort 187
H.4.5.2 Disposal effort 188
H.5 Cost analysis 188
H.5.1 Cost estimating 189
H.5.2 Estimate types 190
H.5.2.1 Life-cycle cost estimate 190
H.5.2.2 Total Ownership Cost 190
H.5.2.3 Analysis of Alternatives 191
H.5.2.4 Independent Cost Estimate 191
H.5.2.5 Program Office Estimate (POE) 191
H.5.2.6 Component Cost Analysis 191
H.5.2.7 Economic Analysis 192
H.6 Acquisition category information 192
H.7 Acquisition references 194
H7.1 Online resources 194
H7.2 Statutory information 194

H7.3 Acquisition decision support systems 198

Appendix I Data Collection 199

I.1 Software data collection overview 199

I.1.1 Model comparisons 199

I.1.2 Format 201

I.1.3 Structure 202

I.2 Software data collection details 202

I.3 CSCI description 202

I.3.1 Requirements 204

I.3.2 Systems integration 204

I.4 Size data 205

I.4.1 Sizing data 205

I.4.1.1 Source code (KSLOC) 205

I.4.1.2 Reuse adjustments 205

I.4.1.3 Software source 206

I.4.1.4 Function points 206

I.4.1.5 Programming source language 206

I.5 Development environment data 206

I.6 Cost, schedule data 207

I.7 Technology constants 208

I.8 Development environment attributes 209

I.8.1 Personnel 209

I.8.2 Support 214

I.8.3 Management 222

I.8.4 Product 225

List of Figures

Figure #	Description	Page
Figure 1-1	Achievable development schedule	5
Figure 1-2	Development environment facets	6
Figure 2-1	Defense acquisition management framework	10
Figure 2-2	Waterfall development	14
Figure 2-3	Software product activities relationship	14
Figure 2-4	Computer software architecture	17
Figure 3-1	Achievable effective size and schedule	24
Figure 3-2	Software elements and relationship to estimate type	26
Figure 4-1	Effective size growth distribution	30
Figure 5-1	Rayleigh-Norden project staffing profile	41
Figure 5-2	Effects of improper staffing	42
Figure 6-1	Source of code taxonomy	50
Figure 6-2	Black box description	51
Figure 6-3	Normal distribution	52
Figure 6-4	Impact of structure on effective size	54
Figure 6-5	Effort required to incorporate changes	55
Figure 6-6	Historic project data basis for growth algorithm	58
Figure 6-7	Modified Holchin growth algorithm	59
Figure 6-8	Effective size growth distribution	62
Figure 6-9	Function point system structure	65
Figure 6-10	State transition model	71
Figure 8-1	Basic technology constant range	86
Figure 8-2	Basic technology constant distribution	87
Figure 8-3	Productivity gains from 1960 to present	87
Figure 8-4	Learning curve impact	90
Figure 8-5	Impact of Application Experience on development effort	91
Figure 8-6	CMMI rating improvement over period 1987 to 2002	92
Figure 8-7	Components of communication	94
Figure 9-1	Learning curve impact	101
Figure 9-2	Impact of programming language experience on development	102
Figure 9-3	Impact of practices and methods experience on development	103
Figure 9-4	Impact of development system experience on development	103
Figure 9-5	Impact of target system experience on development	104
Figure 10-1	Software complexity illustration	109
Figure B-1	Rayleigh staffing profile	126
Figure D-1	Software waterfall process	136
Figure D-2	Spiral development process	137
Figure D-3	Evolutionary development process	138
Figure D-4	Incremental development process	138
Figure D-5	Agile development process	139
Figure H-1	Defense Acquisition System	177
Figure H-2	Acquisition Oversight	179
Figure H-3	Defense Acquisition Management Framework	180
Figure H-4	User Needs Activities	181
Figure H-5	Pre-Systems Acquisition Activity	182
Figure H-6	Systems Acquisition Activity	183
Figure H-7	Production and Deployment Phase	186
Figure H-8	Operations and Support Phase	186
Figure H-9	Life Cycle Cost Composition	187

List of Tables

Table #	Description	Page
Table 3-1	Typical productivity factors by size and software type	22
Table 4-1	System concept information	27
Table 4-2	Stratification of complexity data	29
Table 4-3	Maximum software growth projections as a function of maturity and complexity	31
Table 4-4	Mean software growth projections as a function of maturity and complexity	31
Table 4-5	Typical productivity factors by size and software type	32
Table 4-6	Electronic Systems Center reliability categories	33
Table 4-7	Definition of complexity/reliability categories	33
Table 4-8	Productivities for military applications by category	34
Table 4-9	Total project effort distribution as a function of product size	35
Table 5-1	Computer Software Configuration Item Size Estimates	39
Table 5-2	Total project effort distribution as a function of product size	47
Table 5-3	Approximate total schedule breakdown as a function of product size	48
Table 6-1	Code growth by project phase	59
Table 6-2	Modified Holchin maturity scale	60
Table 6-3	Mean growth factors for normal complexity values as a function of maturity	61
Table 6-4	Maximum growth factors for normal complexity values as a function of maturity	61
Table 6-5	Function point rating elements	65
Table 6-6	Table of weights for function point calculations	67
Table 6-7	Ranking for Internal Logical and External Interface Files	67
Table 6-8	Unadjusted function point calculation	68
Table 6-9	Ranking for External Inputs	68
Table 6-10	Ranking for External Outputs	69
Table 6-11	Ranking for External Inquiries	69
Table 6-12	Ranking for Transforms	70
Table 6-13	Unadjusted function point calculation	71
Table 6-14	General System Characteristics definition	72
Table 6-15	General System Characteristic ratings	72
Table 6-16	Online Data Entry rating definitions	73
Table 6-17	Function Point to Source Lines of Code conversion	74
Table 7-1	Typical productivity factors by size, type, and complexity value	79
Table 7-2	Typical productivity factors by size and software type	79
Table 7-3	Definition of complexity/reliability categories	81
Table 7-4	Productivity for military applications by category	81
Table 7-5	Productivity for military applications as a function of personnel capability	81
Table 7-6	Relationship between C_k and PI values	82
Table 7-7	Typical PI ranges for major application type from the QSM database	83
Table 8-1	Analyst capability ratings	89
Table 8-2	Programmer capability ratings	89
Table 8-3	Traditional use of modern practices rating	91
Table 8-4	Relationship between CMMI and MODP ratings	92
Table 8-5	Modern tool categories and selection criteria	93
Table 8-6	Use of automated tools support rating	93
Table 9-1	Programming language mastery time	102
Table 9-2	Development system volatility ratings	104
Table 9-3	Practices/methods volatility ratings	105
Table 9-4	Multiple security classifications ratings	105
Table 9-5	Multiple development organizations ratings	105
Table 9-6	Multiple development site ratings	106
Table 9-7	Resources and support location ratings	106
Table 10-1	Stratification of complexity data	108
Table 10-2	Complexity rating matrix	110

Table #	Description	Page
Table 10-3	Special display requirements ratings	110
Table 10-4	Rehosting requirements ratings	111
Table 10-5	Memory constraint ratings	111
Table 10-6	Required reliability ratings	112
Table 10-7	Real time operation ratings	113
Table 10-8	Requirements volatility ratings	114
Table 10-9	Security requirements ratings	115
Table E-1	Comparison of major software estimating methods	142
Table E-2	Typical productivity factors by size and software type	147
Table E-3	Environment factors used by common third-order estimation models	150
Table F-1	Baseline description of case study at concept stage	152
Table F-2	Case study unadjusted function point calculation	153
Table F-3	Case study general system characteristics ratings	153
Table F-4	Function point to Source Lines Of Code conversion	154
Table F-5	Baseline description of case study at concept stage	154
Table F-6	Software growth projections as a function of maturity and complexity	155
Table F-7	Baseline description of case study at concept stage	155
Table F-8	Definition of complexity/reliability categories	156
Table F-9	Productivity values for military applications by category	156
Table F-10	Productivity values for case study derived from the ESC database	157
Table F-11	Comparison of cost with mean and maximum size growth using ESC data	157
Table F-12	Comparison of cost with mean and maximum size growth using table	158
Table F-13	Comparison of worst case from component-level and system level	159
Table F-14	Total project effort distribution as a function of product size	159
Table F-15	Total project effort distribution for nominal case study development	160
Table F-16	Maximum effort analysis from system level including maintenance	161
Table G-1	Baseline description of case study at start of requirements review	162
Table G-2	Case study unadjusted function point calculation	163
Table G-3	Case study general system characteristics ratings	164
Table G-4	Baseline description of case study at start of requirements review	165
Table G-5	Software growth projections as a function of maturity and complexity	166
Table G-6	Baseline description of case study at start of requirements development	166
Table G-7	Parameter values for basic capability estimate calculation	167
Table G-8	Personnel parameter values for case study	167
Table G-9	Development environment parameter values for case study	168
Table G-10	Product impact parameter values for case study	168
Table G-11	Technology constant values for case study	169
Table G-12	Nominal effort and schedule analysis of case study at the component level	170
Table G-13	Worst-case effort and schedule analysis of case study at component level	170
Table G-14	Total project effort distribution for case study	172
Table G-15	Nominal effort allocation for the case study at the component level	173
Table G-16	Approximate schedule breakdown as a function of product size	174
Table G-17	Nominal schedule allocation for the case study at the component level	175
Table G-18	Nominal component level cost analysis of case study maintenance	176
Table H-1	DoD Instruction 5000.2 Acquisition Categories	192
Table H-2	SECNAV Instruction 5000.2C Acquisition Categories	193
Table I-1	Estimating model parameter comparison	200
Table I-2	Computer Software Configuration Item information	202
Table I-3	Project summary data	202
Table I-4	Requirements data	204
Table I-5	System integration data	204
Table I-6	Source code sizes	205
Table I-7	Reuse data	206
Table I-8	Reuse source	206
Table I-9	Function point data	206

Table #	Description	Page
Table I-10	Source language	206
Table I-11	Environment data	207
Table I-12	Development effort and schedule data	208
Table I-13	Technology constants	208
Table I-14	Analyst Capability (ACAP) rating values	209
Table I-15	Programmer Capability (PCAP) rating values	210
Table I-16	Productivity Factor (PROFAC) rating values	210
Table I-17	Application Experience (AEXP) rating values	211
Table I-18	Development System Experience (DEXP) rating values	212
Table I-19	Programming Language Experience (LEXP) rating values	212
Table I-20	Practices and Methods Experience (PEXP) rating values	213
Table I-21	Target System Experience (TEXP) rating values	213
Table I-22	Development System Complexity (DSYS) rating values	214
Table I-23	Development System Volatility (DVOL) rating values	215
Table I-24	Modern Practices use (MODP) rating values	216
Table I-25	Process Improvement (PIMP) rating values	217
Table I-26	Practices/methods Volatility (PVOL) rating values	218
Table I-27	Reusability level required (RUSE) rating values	219
Table I-28	Required Schedule (SCED) rating values	219
Table I-29	Automated tool support levels of automation	220
Table I-30	Automated tool use (TOOL) rating values	221
Table I-31	Multiple Classification Levels (MCLS) rating values	222
Table I-32	Multiple Development Organizations (MORG) rating values	223
Table I-33	Multiple Development Sites (MULT) rating values	224
Table I-34	Personnel Continuity (PCON) rating values	224
Table I-35	Product Complexity (CPLX) rating values	225
Table I-36	Database size rating values	226
Table I-37	Special Display requirements (DISP) rating values	227
Table I-38	Development Re-hosting (HOST) rating values	228
Table I-39	External Integration (INTEGE) requirements rating values	229
Table I-40	Internal Integration (INTEGI) requirements rating values	229
Table I-41	Target System Memory Constraints (MEMC) rating values	230
Table I-42	Software Platform (PLAT) rating values	230
Table I-43	Required Software Reliability (RELY) rating values	231
Table I-44	Real-time operations requirements (RTIM) rating values	232
Table I-45	System Requirements Volatility (RVOL) rating values	232
Table I-46	System Security Requirement (SECR) rating values	233
Table I-47	System CPU Timing Constraint (TIMC) rating values	235
Table I-48	Target System Volatility (TVOL) rating values	236

List of Equations

Equation #	Description	Page
Equation 3-1	First-order estimating model	21
Equation 3-2	Simple size value	22
Equation 3-3	Component-level estimating model	25
Equation 3-4	Single CSCI development schedule approximation	26
Equation 4-1	System-level estimating model	27
Equation 4-2	Single CSCI development schedule approximation	27
Equation 4-3	Total effort relationship	35
Equation 4-4	Enhancement effort component	37
Equation 4-5	Knowledge retention effort heuristic	37
Equation 4-6	Steady-state maintenance effort	38
Equation 5-1	Component-level development effort	39
Equation 5-2	Component development schedule	40
Equation 5-3	Rayleigh-Norden staffing profile relationship	42
Equation 5-4	Basic technology constant	44
Equation 5-5	Effective technology constant	45
Equation 5-6	General component-level effort	45
Equation 5-7	Development schedule	46
Equation 5-8	Alternate schedule equation	46
Equation 5-9	Total effort	47
Equation 6-1	Mean size	53
Equation 6-2	Standard deviation	53
Equation 6-3	Adaptation adjustment factor	56
Equation 6-4	Size adjustment factor	56
Equation 6-5	Effective size (Jensen-based)	56
Equation 6-6	Effective size (COCOMO-based)	57
Equation 6-7	Ratio of development cost to development time	59
Equation 6-8	Growth relationship (minimum)	60
Equation 6-9	Growth relationship (maximum)	60
Equation 6-10	Effective size growth	60
Equation 6-11	Total size growth	61
Equation 6-12	Effective size	61
Equation 6-13	Effective size growth	62
Equation 6-14	Mean growth size	63
Equation 6-15	Value adjustment factor	73
Equation 6-16	Adjusted function point count	73
Equation 6-17	Function points to source lines of code conversion	74
Equation 6-18	Effective size	75
Equation 7-1	Development effort	77
Equation 7-2	Hours per source line of code	77
Equation 7-3	General form of SLIM® software equation	82
Equation 7-4	Productivity factor relationship	82
Equation 7-5	Development effort for upgrade	83
Equation 7-6	Development schedule	84
Equation 8-1	Basic technology constant	94
Equation 8-2	Productivity factor	94
Equation 10-1	Complexity function	108
Equation E-1	First-order estimating model	147
Equation E-2	Effective size	147
Equation E-3	Second-order estimating model	148
Equation E-4	Effective size	148
Equation E-5	Third-order estimating model	149
Equation F-1	Function point value adjustment factor	153

Equation #	Description	Page
Equation F-2	Adjusted function point count	154
Equation F-3	Total software source lines of code count	154
Equation F-4	Total effort relationship	159
Equation G-1	Function point value adjustment factor	164
Equation G-2	Adjusted function point count	164
Equation G-3	Total software source lines of code count	164
Equation G-4	Mean size growth factor	166
Equation G-5	Maximum growth size	166
Equation G-6	Productivity factor	169
Equation G-7	Median productivity factor	169
Equation G-8	Total effort	173
Equation I-1	Development effort	200
Equation I-2	Development productivity	206

Acknowledgements

The size of this guidebook hardly represents the amount of effort expended to develop it. Since May 2005, nearly five person years have gone into the research, writing, editing, and production.

We wish to thank Susan Wileman of the Naval Center for Cost Analysis (NCCA), specifically for her confidence in our experience and knowledge and her support over the years during the development of the guidebook.

We appreciate Susan, John Moskowitz, Mike Tran, and others for the reviews, discussions, and feedback during the many months of writing. We offer our gratitude to members of the United States Air Force Software Technology Support Center (STSC) for their expert editing and proof-reading: Thomas Rodgers, Gabriel Mata, Daniel Keth, Glen Luke, and Jennifer Clement. We offer a special thanks to Dr. David A. Cook, Dr. James Skinner, and Teresa Brown for their singular knowledge and perspective in the technical review. The final review conducted by the CrossTalk editors, namely Drew Brown and Chelene Fortier-Lozancich, uncovered a myriad of fine points we overlooked or assumed everyone would know or understand.

Most significantly, we acknowledge the extensive knowledge and experience of Dr. Randall W. Jensen applied in writing and refining this guidebook into what, arguably, will become a standard within the cost estimating community. Additional writing, editing, and final production by Leslie (Les) Dupaix and Mark Woolsey were also key efforts in creating this volume. We also wish to acknowledge the Defense Acquisition University for providing the basis of the acquisition related portions of the guidebook.

Finally, we want to thank our spouses, families, and anyone else, who would listen to our repetitive discussions on topics within the guidebook, endured us working on it at home or during vacations, and were always interested (or at least pretended to be) in our efforts to create a useful resource for software cost estimators.

Predicting is very hard, especially when it is about the future.

Yogi Berra

Section 1

Introduction

The term "software crisis" refers to a set of problems that highlight the need for changes in our existing approach to software development. The term "software crisis" originated sometime in the late 1960s about the time of the 1968 NATO Conference on Software Engineering. One of the most dominant and serious complaints arising from the software crisis was the inability to estimate, with acceptable accuracy, the cost, resources, and schedule required for a software development project.

Crisis is a strong word. It suggests a situation that demands resolution. The conditions that represent the crisis will be altered, either toward favorable relief or toward a potential disaster. According to Webster's definition, a crisis is "a crucial or decisive point or situation." By now, the crisis should have been resolved one way or another.

A notion pervading the conference was that we can engineer ourselves out of any problem. Hence, the term "software engineering" was coined. One of the significant conference outputs was a software engineering curriculum. The curriculum happened to be identical to the computer science curriculum of that day.

A list of software problems was presented as major development concerns at the 1968 NATO Conference. The problem list included software that was:

- Unreliable
- Delivered late
- Prohibitive in terms of modification costs
- Impossible to maintain
- Performing at an inadequate level
- Exceeding budget costs

The software development problems listed in 1968 are still with us today. Each of these complaints can be traced to the inability to correctly estimate development costs and schedule. Traditional intuitive estimation methods have consistently produced optimistic results which contribute to the all too familiar cost overruns and schedule slips. In retrospect, the term *exigence*[1] fits the situation better than "crisis" since there is no discernable point of change for better or worse.

Most humans, especially software developers, are inherent optimists. When was the last time you heard something like, "It can't be that bad," "It shouldn't take more than two weeks to finish," or, best of all, "We are 90 percent complete?" Estimates need to be based on facts (data), not warm feelings or wishful thinking. In other words, hope is not a management strategy, nor is it an estimating approach.

> *…It was also becoming painfully evident that estimating the cost of technologically state-of-the-art projects was an inexact science. The experts, in spite of their mountains of numbers, seemingly used an approach descended from the technique widely used to weigh hogs in Texas. It is alleged that in this process, after catching the hog and tying it to one end of a teeter-totter arrangement, everyone searches for a stone which, when placed on the other end of the apparatus, exactly balances the weight of the hog. When such a stone is eventually found, everyone gathers around and tries to guess the weight of the stone. Such is the science of cost estimating. But then, economics has always been known as the dismal science.*
>
> Augustine's Laws

[1] Exigence: The state of being urgent or pressing; urgent demand; urgency; a pressing necessity.

The cost and schedule estimating problem can be described by the following statement:

> *More software projects have gone awry for lack of calendar time than for all other causes combined. Why is this cause of disaster so common?*
>
> *First, our techniques of estimating are poorly developed. More seriously, they reflect an unvoiced assumption which is quite untrue, i.e., that all will go well.*
>
> *Second, our estimating techniques fallaciously confuse effort with progress, hiding the assumption that men and months are interchangeable.*
>
> *Third, because we are uncertain of our estimates, software managers often lack the courteous stubbornness of Antoine's chef.*
>
> *Fourth, schedule progress is poorly monitored. Techniques proven and routine in other engineering disciplines are considered radical innovations in software engineering.*
>
> *Fifth, when schedule slippage is recognized, the natural (and traditional) response is to add manpower. 'Like dousing a fire with gasoline, this makes matters worse, much worse. More fire requires more gasoline and thus begins a regenerative cycle that ends in disaster.'*[2]

Antoine's is a New Orleans restaurant whose menu states: good cooking takes time. If you are made to wait, it is to serve you better and to please you.

The rapidly increasing cost of software has led customers for these products to become less willing to tolerate the uncertainty and losses associated with inaccurate cost and schedule estimates, unless the developer is willing to accept a significant portion of that risk. This customer pressure emphasizes the need to use an estimating method that can be applied early in the software development when tradeoff studies and investment decisions are made. The estimating method must consider the characteristics of the development organization and the environmental effects imposed by the development task, as well as the application size and complexity.

Estimating is magic for most estimators and managers. Well-known science fiction author Arthur C. Clarke's Third Law[3] states: "Any sufficiently advanced technology is indistinguishable from magic." This illustrates one of the primary problems with software estimating today. The "magic" creates an environment of unreasonable trust in the estimate and lack of rational thought, logical or otherwise.

es·ti·mate

To make a judgment as to the likely or approximate cost, quality, or extent of; calculate approximately…estimate may imply judgment based on rather rough calculations.

American Heritage Dictionary

Estimating tools produce estimates from a set of inputs that describe the software and the environment. The result is a development cost and schedule. If neither the estimator, nor the manager, understands the algorithm behind the estimate, then the estimate has crossed into the realm of magic. The result is a development cost and schedule estimate – which is never wrong. Estimate accuracy appears to increase with the number of significant digits.

With magic we expect the impossible, and so it is with estimating as well. When something is magic, we don't expect it to follow logic, and we don't apply our common sense. When the estimate is not the cost and schedule we

---

[2] Brooks, F.P. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, Reading, MA: 1975.
[3] Clarke, Arthur C. Clarke. Profiles of the Future. 1961.

want, we can simply change the inputs to the algorithm and produce the estimate we desire. Since the tool has magical properties, we can suspend reality and make any estimate come true. That is why so many projects overrun and we consistently blame the failure on the projects, not the estimates.

As demonstrated, software cost estimation is a discipline sometimes equated with mystical forms of prognostication, in which signs and indications are used to foretell or predict the future. Some suggest that a venerable person of recognized experience and foresight with far-seeing wisdom and prudence is required to create accurate cost estimates. These points have merit as cost estimation may fittingly be considered a blend of art and science. Yet, with proper knowledge and experience wisely applied, the discipline of software cost estimating becomes more science than knack.

Several cost and schedule estimation methods have been proposed over the last 25 years with mixed success due, in part, to limitations of the estimation models. A significant part of the estimate failures can be attributed to a lack of understanding of the software development environment and the impact of that environment on the development schedule and cost. The environment imposed by the project manager is a major driver in the software equation.

Organizations need both managers and estimators. Managers make infrequent estimates to support their decisions. Estimators, like any specialist, need to perform frequent estimates to track development progress, increase estimating experience, and maintain process and tool proficiency.

Software cost estimating is an essential part of any system acquisition process. Fiscal constraints, the mission of service-level and command cost agencies, and program office responsibilities further highlight the importance of a solid understanding of software cost estimating principles and the need for credible resources.

Estimation seeks to answer questions such as:

- Is the estimate reasonable?
- Has a similar system been developed before?
- How long should the development take?
- How much should the development cost?
- How does size, cost, and schedule data from other projects relate to this system?
- If developers claim their software development productivity is 1,000 lines of source code per month, are the claims realistic?

> *Programming a computer does require intelligence. Indeed, it requires so much intelligence that nobody really does it very well. Sure, some programmers are better than others, but we all bump and crash around like overgrown infants. Why? Because programming computers is by far the hardest intellectual task that human beings have ever tried to do.* ***Ever****.*
>
> G.M. Weinberg, 1988

A sound software cost estimate is developed by employing recognized and accepted estimating methodologies. As an analyst, you can determine that your estimate is credible by using historical data, cost estimating relationships, and having an understanding of the tools or models used.

Estimating the cost, schedule and resources for a software development project requires training, experience, access to historical information related to the software domain under consideration, and the confidence to commit to the estimate even when the project information is qualitative and lacks detail. All software estimates carry inherent risks from several views. For example, all estimating tools are the results of regression analysis (curve fitting) to historic project data that is inconsistent in nature. Data is collected from

many sources, each with its own definition of size, complexity, productivity, and so on. Knowledge of the software project is somewhat subjective in terms of size, complexity, the environment, and the capabilities of the personnel working on project development.

Risk represents the degree of uncertainty in the cost and schedule estimates. If the scope of the system under development is poorly understood, or the software requirements are not firm, uncertainty can become extreme. Software requirements in an ideal world should be complete and specified at a level that is sufficient to the maturity of the system. Interfaces should also be complete and stable to reduce the instability of the software development and the development estimate.

1.1 Development constraints

There is a model of software development as seen from the project control point of view. This model has only four variables:

- Cost
- Schedule
- Quality
- Scope

Project Uncertainty Principle

If you understand a project, you won't know its cost, and vice versa.

Dilbert (Scott Adams)

The shareholders (users, customers, etc. – all external to the development) are allowed to set three of the four variables; the value of the fourth variable will be determined by the other three.

Some managers attempt to set all four variables, which is a violation of the rules. When one attempts to set all four, the first visible failure is a decrease in product quality. Cost and schedule will then increase in spite of our most determined efforts to control them. If we choose to control cost and schedule, quality and/or scope become dependent variables.

The values for these attributes cannot be set arbitrarily. For any given project, the range of each value is constrained. If any one of the values is outside the reasonable range, the project is out of control. For example, if the scope (size) is fixed, there is a minimum development time that must be satisfied to maintain that scope. Increasing funding to decrease the schedule will actually increase the schedule while increasing the project cost.

Software estimating tools allow us to make the four variables visible so we can compare the result of controlling any, or all, of the four variables (look at them as constraints) and their effect on the product.

1.2 Major cost factors

There are four major groups of factors that must be considered or accounted for when developing a software cost and schedule estimate. The factor groups are: (1) effective size, (2) product complexity,
(3) development environment, and (4) product characteristics.

The following sections briefly describe each of the major factor groups.

1.2.1 Effective size

Effective size is a quantification of the effort required to produce a software product. It does not directly correlate with the physical product size. Software size is the most important cost and schedule driver, yet it is the most difficult to determine. Size prediction is difficult enough that many methods have been created to alleviate the problem. These measures include effective source lines of code (ESLOC), function points, object points, and use cases, as well as many variants.

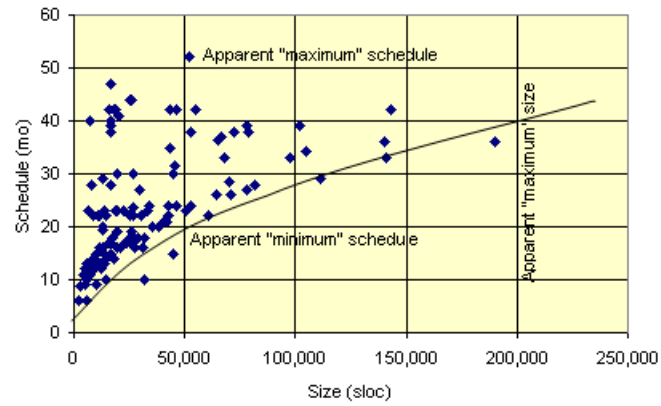


Figure 1-1: Achievable development schedule based on 130 military software projects

Size has a major impact on the management of software development in terms of cost and schedule. An Aerospace Corporation study by Long et al[4] that examined 130 military software development projects demonstrates some of the constraints on development imposed by the magnitude of the software project. Of the 130 projects shown in Figure 1-1, no Computer Software Configuration Item (CSCI) over 200 thousand effective source lines of code (KESLOC) was successfully completed or delivered. A project effective size of 200 KESLOC requires a team of approximately 100 development and test personnel, a development schedule of four years, and nearly 3,000 person months of effort. The average turnover rate for software personnel is less than four years. Managing a team of 100 people in a single development area is not trivial.

The Aerospace study also showed that schedule is not arbitrary. There is an apparent minimum development schedule related to size. The assumption that by front-loading project staff the project will decrease its schedule below a minimum development time is faulty. This is sometimes referred to as the software Paul Masson Rule; that is, "We will deliver no software before its time."

> *We will sell no wine before its time.*
>
> Paul Masson advertisement, 1980

Effective software size is discussed in detail in Section 6.

1.2.2 Product complexity

Software complexity is an all-embracing notion referring to factors that decide the level of difficulty in developing software projects. There are many facets to the value we refer to as complexity. Here we will only touch upon the effects of product complexity and its importance in cost and schedule estimation. First, complexity limits the rate at which a project can absorb development personnel. It also limits the total number of people that can effectively work on the product development. Small development teams are actually more productive per person than large teams; hence, the limiting action of complexity correlates with higher productivity. At the same time, the skill set necessary to build an operating system is not interchangeable with that of a payroll system developer.

Complexity is discussed in detail in Section 10.

[4] Long, L., K. Bell, J. Gayek, and R. Larson. "Software Cost and Productivity Model." *Aerospace Report No. ATR-2004(8311)-1*. Aerospace Corporation. El Segundo, CA: 20 Feb 2004.

1.2.3 Development environment

The development environment is a major, yet often ignored, factor in development productivity and the resulting development cost and schedule. The environment may be divided into two areas: developer capability and project-specific development environment. At the core of the development environment measure is the raw capability of the developer. This includes application experience, creativity, ability to work as a team, and the use of modern practices and tools. It is hard to directly assess the capability or the environment, but by looking at the individual, independent facets of the environment (shown in Figure 1-2), a cohesive, rational measure can be obtained.

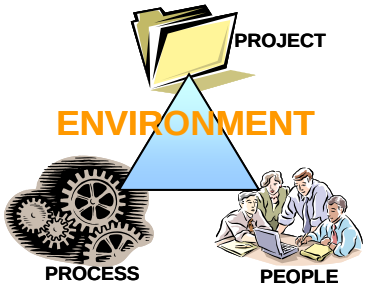


Figure 1-2: Development environment facets

The development environment is discussed in detail in Sections 8 (developer capability) and 9 (development environment).

For estimates conducted early in the acquisition – when the developer and environment are unknown – typical productivity factors that assume a generic developer and environment can be used to obtain a "ballpark" estimate. This technique is described in Section 7.

1.2.4 Product characteristics

Product characteristics describe the software product to be developed. These characteristics are specific to a class of products; for example, a military space payload. The characteristics include timing and memory constraints, amount of real-time processing, requirements volatility, user interface complexity, and development standards (among others). Development standards, in turn, encompass the documentation, reliability, and test requirements characteristics of a project.

The product characteristics generally limit or reduce the development productivity. The characteristics for any application can be grouped into a product template that simplifies a system-level estimate.

The elements of the product characteristics are discussed in detail in Section 10.

1.3 Software support estimation

Software support costs usually exceed software development costs, primarily because software support costs involve much more than the cost of correcting errors. Typically, the software product development cycle spans one to four years, while the maintenance phase spans an additional five to 15 years for many programs. Over time, as software programs change, the software complexity increases (architecture and structure deteriorate) unless specific effort is undertaken to mitigate deterioration[5].

Murphy's Law is an adage that broadly states that *things will go wrong in any given situation, if you give them a chance*. "If there's more than one possible outcome of a job or task, and one of those outcomes will result in disaster or an undesirable consequence, then somebody will do it that way."

Software maintenance cost estimates in 1976 ranged from 50 to 75 percent of the overall software life-cycle costs[6]. The trend is about the same today in spite of the rapid increase in product size over time. Historic project data[7] shows that a program with a software development cost of about $100 per

[5] Belady, L.M., and M.M. Lehman. "Characteristics of Large Systems." Research Directions in Software Technology. MIT Press. Cambridge, MA: 1979.
[6] Boehm, B.W. "Software Engineering," *IEEE Transactions of Computers.* Dec.1976: 1226-1241.
[7] Ibid., Boehm, B.W.

source line can have long term maintenance costs that are near $4,000 per source line.

Software maintenance is defined by Dr. Barry Boehm[8] as: "*The process of modifying existing operational software while leaving its primary functions intact*." His definition includes two types of activities: software repair and software enhancement.

Software repair is another way of saying "corrective maintenance" – fixing implementation, processing, and performance related to the specified software requirements. These failures may surface after delivery of the product in operational use. The repair can also be related to errors or deficiencies known before the product was delivered, but deferred to maintenance because of funding or schedule issues during development. Maintenance costs during the period immediately following product delivery are normally high due to errors not discovered during the software development.

Software enhancement results from software requirements changes during and following product delivery. Software enhancement includes:

- Redevelopment or modification of portions of the existing software product.
- Software adaptation ("adaptive") to new processing or data environments.
- Software performance and maintainability enhancements ("perfective").

In addition to repair and enhancement, the cost of maintenance must also include the cost of maintaining knowledge of the software structure and code. The cost of this knowledge retention for a large software system often dominates the maintenance cost.

1.4 Guidebook overview

This guidebook is divided into major sections centered on key software development cost estimating principles. The sections discussed include the following:

- **Software cost and schedule estimating introduction (Sections 1 & 2)** – The history, art, and science behind developing reasonable cost estimates.
- **Software development process (Section 2)** – The evolution of software development, various methods or approaches, and key issues.
- **Levels of detail in software estimates (Section 3)** – Guidelines and theory to help determine the best or most appropriate method for evaluating given data and formulating an estimate. The cost of developing the system is just the tip of the iceberg when the cost over the entire life of the program is considered.
- **System level estimating process (Section 4)** – A concise introduction to the software cost estimating process at the system level, taking place prior to knowledge of the software architecture.

[8] Boehm, B.W. Software Engineering Economics Prentice-Hall. Englewood Cliffs, NJ. 1981: 54.

This process assumes a generic developer and total effective software size, including growth. Estimates include validation and effort allocation.

- **Component level estimating process (Section 5)** – A concise introduction to the software cost and schedule estimating process at the component level using the effective component (CSCI) size including growth, the developer capability and environment, and product constraints to determine the development cost and schedule. Estimates include cost and schedule validation, as well as effort and schedule allocation.
- **Estimating effective size (Section 6)** – A key element in determining the effort and subsequent cost and schedule is the size of the software program(s) within the system. This section explains two primary methods of size estimation: effective source lines of code and function points.
- **Productivity factor evaluation (Section 7)** – Software development effort estimates at the system level are dependent upon the effective size and the generic developer productivity for the given system type. Productivity factors can be derived from historic industry data or from specific developer data (if available).
- **Evaluating developer capability (Section 8)** – An important factor in a component level estimate is the developer's capability, experience, and specific qualifications for the target software system. This section explains the attributes that define developer capabilities in terms of efficiency or productivity.
- **Development environment evaluation (Section 9)** – The development environment (management, work atmosphere, etc.) and product traits (complexity, language, etc.), combined with the developer's skill and experience directly impact the cost of the development. This section quantitatively describes the impacts of the development environment in terms of productivity.
- **Product characteristics evaluation (Section 10)** – The product characteristics (real-time operation, security, etc.) and constraints (requirements stability, memory, CPU, etc.) reduce the development efficiency and increase cost and schedule. This section quantitatively describes the impacts of the product characteristics on development cost and schedule and the associated estimates.
- **Acronyms (Appendix A)** – This appendix contains a list of the acronyms used in the guidebook and by the estimating tools supported by the guidebook.
- **Terminology (Appendix B)** – This section contains definitions and terminology common to this guidebook and the software estimating discipline.
- **Bibliography (Appendix C)** – This section contains a list of useful resources for software development cost and schedule estimators. The list also includes material useful for software development planners and managers.
- **Software life cycle approaches (Appendix D)** – This appendix provides background information describing the common software

development approaches and discusses topics such as spiral and incremental development.

- **Software estimating models (Appendix E) –** This section describes the myriad of estimate types and models available for the software cost and schedule estimator.
- **System-level estimate case study (Appendix F**) **–** This section walks the reader through a system-level estimate example, incorporating the processes, practices, and background material introduced in the core sections of the guidebook.
- **Component-level estimate case study (Appendix G) –** This section, like the previous section, works through an in-depth example at the component level.
- **The Defense Acquisition System (Appendix H) –** This appendix provides background information describing the defense acquisition framework and how software estimates fit into that process.
- **Data collection (Appendix I) –** Data collection (as highlighted in the core sections) without efforts to validate and normalize it is of limited value. This section provides guidelines, instructions, and suggested formats for collecting useful data.

Researchers have already cast much darkness on the subject and if they continue their investigations, we shall soon know nothing at all about it.

Mark Twain

Section 2

Software Development Process

The phrase "software life cycle" became popular in the 1970s. The phrase caught on and has remained in use because it conveys an impression of multiple phases and extended life in the software development process. The previously popular conception of computer programs as items that are developed inside someone's head, coded, and used for a day or 10 years without change has frustrated most people in the industry at one time or another. The facts are, and have been for some time, that coding represents only a small percentage of the money and effort spent on a typical software system.

One often-used effort division model assumes percentage split of 40/20/40 for the analysis/design-code-test/integration phases, respectively. Most modern software estimating tools assume an effort division percentage of 40/20/40 or 40/30/30. If the maintenance phase is included, the relative percentage for coding drops to an even lower value because maintenance cost/time tends to exceed any other phase. Thus, the concept of a life cycle is welcome and realistic because it calls attention to phases other than coding and emphasizes the fact that software systems "live" a long, long time.

There are many representations of a life cycle. Each software industry subculture has its own – or several – representations, and each representation tends to be modified somewhat for specific projects. There are many generic approaches that characterize the software development process. Some of the approaches (process models) that are in use today are described in Appendix D. A generic engineering process, known as the waterfall model, is used in this section to describe the activities in the typical process.

IEEE 12207

An Institute of Electrical and Electronics Engineers standard that establishes a common framework for the software life cycle process.

This officially replaced MIL-STD-498 for the development of DoD software systems in August 1998.

Cost:
The expenditure of something, such as time or labor, necessary for the attainment of a goal.

Schedule:
A plan for performing work or achieving an objective, specifying the order and allotted time for each part.

2.1 The Defense Acquisition System

The software development process fits into a larger acquisition process referred to as the Defense Acquisition Management Framework[9], which is a continuum of activities that represent or describe acquisition programs. The framework is represented by Figure 2-1.

2.1.1 Framework elements

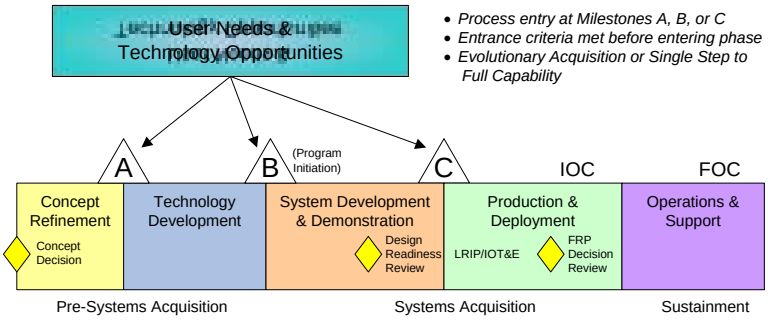


Figure 2-1: Defense Acquisition Management Framework

[9] DoD Instruction 5000.2. "Operation of the Defense Acquisition System." May 12, 2003.

The Acquisition Management Framework is separated into three activities: Pre-Systems Acquisition, Systems Acquisition, and Sustainment. These activities are divided into five phases: Concept Refinement, Technology Development, System Development & Demonstration, Production & Deployment, and Operations & Support. The Phases in System Acquisition and Sustainment are further divided into six efforts: System Integration, System Demonstration, Low-Rate Initial Production (LRIP) or limited deployment (if applicable), Full-Rate Production (FRP) and Deployment, Sustainment, and Disposal.

A detailed explanation of the Defense Acquisition System and the Acquisition Management Framework can be found in Appendix H.

The framework figure indicates key points in the process known as milestones. A Milestone is the point at which a recommendation is made and approval sought regarding starting or continuing an acquisition program (i.e., proceeding to the next phase). The milestones established by DoDI 5000.2 are: Milestone A approves entry into the Technology Development phase; Milestone B approves entry into the System Development and Demonstration phase; and Milestone C approves entry into the Production and Deployment phase. Also of note are the Concept Decision that approves entry into the Concept Refinement phase; the Design Readiness Review that ends the System Integration effort and continues the System Development and Demonstration phase into the System Demonstration effort; and the Full Rate Production Decision Review at the end of the Low Rate Initial Production effort of the Production & Deployment phase that authorizes Full Rate Production and approves deployment of the system to the field or fleet.

2.1.2 User needs and technology opportunities

The user needs and technology opportunities effort is divided into two primary areas, User needs activities and technology assessment. User need activities consist of determining the desired capabilities or requirements of the system and is governed by the Initial Capabilities Document (ICD). The user needs activities also involve preparation of the Capability Description Document (CDD) and the Capability Production Document (CPD). Technology assessment examines multiple concepts and materiel approaches to optimize the way the Department of Defense provides these capabilities. The examination includes robust analyses that consider affordability, technology maturity, and responsiveness.

2.1.3 Pre-systems acquisition

Pre-systems acquisition activities involve development of user needs, science and technology efforts, and concept refinement work specific to the development of a materiel solution to an identified, validated need. The activities are governed by the ICD, the CDD and the CPD. The ICD, the CDD and the CPD are created by the program acquisition office in conjunction with “representatives from multiple DoD communities” that assist in formulating broad, time-phased, operational goals, and describing requisite capabilities.

2.1.3.1 Concept refinement phase

Concept refinement begins with the Concept Decision. Entrance into the Concept refinement phase depends upon an approved Initial Capabilities

Document and an approved plan for conducting an Analysis of Alternatives (AoA) for the selected concept. The AoA is created by the program acquisition office. The phase ends when the Milestone Decision Authority approves the preferred solution resulting from the Analysis of Alternatives and approves the associated Technology Development Strategy.

2.1.3.2 Milestone A

At this milestone, the Milestone Decision Authority approves the Technology Development Strategy and sets criteria for the Technology Development phase. If there is no predetermined concept and evaluation of multiple concepts is needed, the Milestone Decision Authority will approve a Concept Exploration effort. If there is an acceptable concept without defined system architecture, the Milestone Decision Authority will approve a Component Advanced Development effort.

2.1.3.3 Technology development phase

During the Technology Development phase, the Capability Development Document is created. This document builds upon the Interface Control Document and provides the necessary details to design the proposed system. The project exits the phase when an affordable increment of useful capability to the military has been identified, demonstrated in the relevant environment, and can be developed within a short timeframe (usually less than five years).

2.1.3.4 Milestone B

The purpose of Milestone B is to authorize entry into the System Development & Demonstration phase. Milestone B approval can lead to either System Integration or System Demonstration depending upon the maturity of the technology.

2.1.4 Systems acquisition

The activity is divided into two phases: System Development & Demonstration and Production & Deployment.

2.1.4.1 System development & demonstration

The objective of the System Development and Demonstration phase is to demonstrate an affordable, supportable, interoperable, and producible system in its intended environment. The phase has two major efforts: System Integration and System Demonstration. System Integration requires a technical solution for the system. At this point, the subsystems are integrated, the detailed design is completed, and efforts are made to reduce system-level risks. The effort is guided by an approved Capability Development Document which includes a minimum set of Key Performance Parameters (KPPs). The program exits System Integration when the system has been demonstrated using prototype articles or Engineering Design Models and upon completion of a Design Readiness Review. Completion of this phase is dependent upon a decision by the Milestone Decision Authority to commit to the program at Milestone C or to end the effort.

2.1.4.2 Milestone C

The purpose of Milestone C is to authorize entry into: Low Rate Initial Production, production or procurement (for systems that do not require Low Rate Initial Production), or limited deployment for Major Automated

Information Systems or software-intensive systems with no production components. A favorable Milestone C decision commits the DoD to production of the system. At Milestone C, the Milestone Decision Authority approves: the updated acquisition strategy, updated development Acquisition Program Baseline, exit criteria for Low Rate Initial Production or limited deployment, and the Acquisition Decision Memorandum (ADM[10]).

2.1.4.3 Production & deployment

The purpose of the Production and Deployment phase is to achieve an operational capability that satisfies mission needs. The phase consists of the Low Rate Initial Production effort, the Full-Rate Production decision review, and the Full-Rate Production & Deployment effort. The decision to continue to full-rate production, or limited deployment for Major Automated Information Systems or software-intensive systems, requires completion of Initial Operational Test and Evaluation and approval of the Milestone Decision Authority.

2.1.5 Sustainment

The Sustainment activity has one phase, Operations & Support, consisting of two major efforts: Sustainment and Disposal. The objective of this activity is to provide cost-effective support for the operational system over its total life cycle. Operations and Support begins when the first systems are deployed. Since deployment begins in the latter portions of the Production & Deployment phase, these two activities overlap. Later, when a system has reached the end of its useful life, it needs to be disposed of appropriately.

2.2 Waterfall model

The waterfall model is the fundamental basis of most software development approaches and serves well as a basis for describing the typical software development processes. Note the word *typical* allows us to establish steps in a process that are not strictly defined or dictated by DoD standards or specifications. This section outlines steps that are part of the *normal* engineering approach to solving problems and are followed in the generalized waterfall development approach common to system development (as shown in Figure 2-2). The products of each step of the development process are indicated. These products are key elements in supporting the product release and a means of measuring the current position in the process.

[10] The Acquisition Decision Memorandum (ADM) is not limited to Milestone C but can be submitted during many points of the acquisition phase.

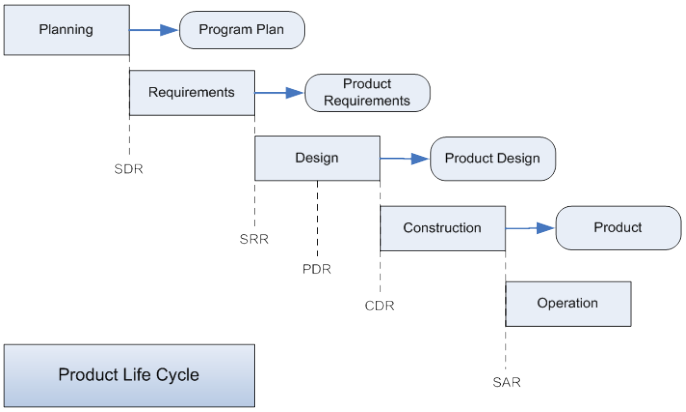


Figure 2-2: Waterfall development cycle

An alternate (and perhaps more revealing) way of presenting the steps or phases of the software development process is by means of the Software V-chart shown in Figure 2-3. The V-chart presents the traditional waterfall process in more detail showing the steps and their relationship to each other.

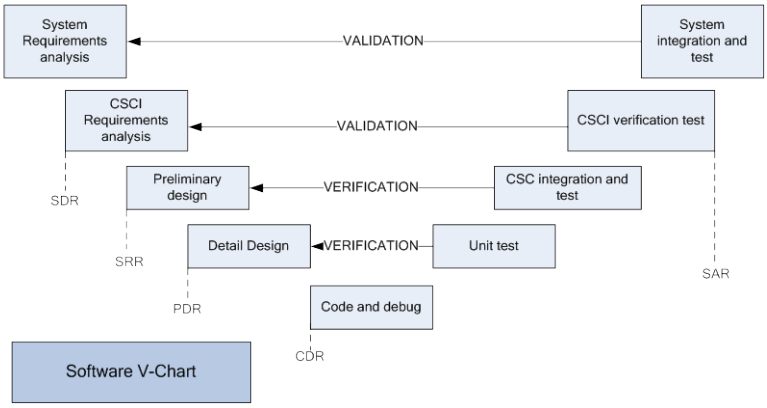


Figure 2-3: Software product activities relationship to integration and test activities

Each step concludes with a formal review to determine the readiness to proceed to the next process step. The requirements activity concludes with a decision point at the Software Requirements Review (SRR) to determine readiness to proceed to full-scale development and begin the high-level or architectural design. The architectural design is evaluated at the Preliminary Design Review (PDR) to determine readiness to proceed to the software detail design. The detail design is reviewed at the Critical Design Review (CDR) to determine readiness to move to the construction phase where the software product is implemented and tested. Once the construction of the software product is complete (computer software units [CSUs]), assembled into Computer Software Components (CSCs), and components are assembled into a product (CSCI), the software product is formally reviewed and accepted at the Software Acceptance Review (SAR) and delivered to the system integration activity.

This is probably the best point to introduce two terms that cause continuing confusion – *verification* and *validation* (V&V). Most people can't tell the difference between V&V. Verification refers to a set of activities that ensure a product is implemented according to its specification, or design baseline. Validation refers to a set of activities that ensures the product satisfies its performance requirements established by the stakeholder. Boehm[11] simplified the definitions:

> *Verification*: "Are we building the product right?"
> *Validation*: "Are we building the right product?"

The *verification* arrows in Figure 2-3 involve testing to ensure performance meets the software requirements. The *validation* arrows at the top of the V assure the product meets stakeholder requirements.

System integration is often beyond the scope of the software development estimate and is included in the system (hardware and software) estimate. However, the software maintenance estimate is typically part of the software estimate.

Developing the software requirements is the first step in software development. (Note that this critical first step is beyond the scope of the software cost and schedule analysis and is included to define the source of software requirements.) The system requirements are developed and a portion (often large) of those requirements is allocated to software prior to the System Design Review (SDR). The SDR decision reviews the system requirements and the allocation of those requirements to hardware, software, and user interaction. The SDR also determines the readiness to move to the next step – the development of the software product.

The steps in the *software* waterfall process are:

- Planning (software requirements analysis).
- Requirements (CSCI requirements analysis and specification).
- Full scale development.
  - Preliminary design (architecture design to PDR).
  - Detail design (PDR to CDR).
  - Construction (code and unit [CSU] development and test).
  - Construction (component [CSC] integration and test).
  - Construction (configuration item [CSCI] integration and test).
- System integration and test.
- Maintenance.

> *The most likely way for the world to be destroyed, most experts argue, is by accident. That's where we come in; we're computer professionals. We cause accidents.*
>
> Nathaniel Borenstein

The following paragraphs contain a more detailed description of the steps in the software waterfall development process shown previously in Figure 2-2. These activities are important and are present in every development approach, whether in the waterfall approach or not. The waterfall approach has its critics, but, as will be seen, the process appears in one form or another in almost every development approach.

2.2.1 Requirements analysis and specification

The second step in the system development process (the first step in the software portion of the process) is to analyze the functions the software subsystem(s) will perform and to allocate those functions to individual CSCIs. The software development team completes the engineering

[11] Boehm, B.W. Software Engineering Economics. Prentice-Hall, Inc. Englewood Cliffs, NJ: 1981.

requirements for each CSCI and the required interfaces. The requirements analysis and specification culminate in a SRR. A complete list of qualification and test requirements are also developed in this activity.

2.2.2 Full-Scale Development

Full-scale software development in the estimator's world is represented by the Figure 2-2 design and construction activities. The completed CSCI is reviewed during the SAR at the conclusion of CSCI integration and test. Successful conclusion of the SAR results in acceptance of the software product by the customer or the system integration activity. The individual phases of the full-scale development are described in the following paragraphs.

The preliminary design creates the software product architecture. The architecture defines each of the CSCs, allocates requirements to each CSC, and defines the interfaces between the CSCs and the external system. The activity culminates in a PDR to determine the readiness for advancement to the detail design.

The detail design allocates requirements from each CSC and establishes the design requirements for each CSU. The activity produces a design for each CSU as well as internal interfaces between CSUs and interfaces to the higher level CSCI. Formal qualification test planning continues and specific test cases are developed. The activity concludes with a formal CDR which reviews the design, the test plans, and critical design issues that have arisen during the activity. Successful completion of the CDR is normally considered the "go-ahead" decision for software production.

> *As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.*
>
> Maurice Wilkes

Software production "turns the corner" from design activities to integration and testing act ivies as shown in Figure 2-3. During the code and unit test activity, Software Product Specifications (SPSs) are drafted and the production code is written for each CSU. The deliverable source code is thoroughly tested since this is the one level that allows direct access to the code. When the unit testing is successfully completed, the CSU is made available for integration with the parent CSC.

Individual CSUs are then assembled into CSCs and tested at the component level. Testing at this level is comprehensive and detailed because this is the last opportunity to stress test the components (i.e., arrays out of bounds, numbers out of range or negative when not expected, invalid inputs, and so forth). When testing shows the CSC functions are performing correctly, the CSCs are integrated into the product-level CSCI.

CSC integration is the final activity in the full-scale CSCI development. Integration testing verifies performance of the entire software product (CSCI) against the formal software requirements approved at the SRR. The test is normally preceded by one or more Test Readiness Reviews (TRRs) to determine if the CSCI is ready for the Formal Qualification Test (FQT). A Formal Qualification Audit (FQA) verifies that all requirements have been incorporated into the design and each requirement has been adequately tested.

At the successful conclusion of the CSCI acceptance test, a formal SAR is conducted prior to acceptance by the customer or the next higher level system integration activity.

2.2.3 System Integration and Test

The final step in the system development process is system integration and test. At this point in the development cycle, the software and hardware configuration items are integrated into the final configuration. This includes user (manuals) operations to determine if the system satisfies the system level requirements before deployment in the operational phase. A full description of system integration and testing is beyond the scope of this guidebook.

2.3 Software development products

It is important to review some software product definitions that are used throughout the guidebook. Important terms describing the product are graphically presented in Figure 2-4. The software product can be a standalone system or a subsystem contained within a larger system structure. In any case, the product is the root of the development structure (project). The product consists of one or more Computer Software Configuration Items (CSCIs). The CSCI relates directly to the allocated software requirements. The CSCI is the component normally used in the detailed software development cost and schedule estimates.

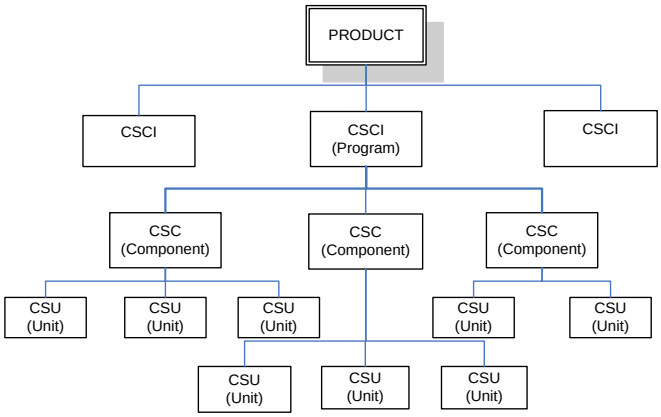


Figure 2-4: Computer software architecture

The CSCI is generally constructed from one or more Computer Software Components (CSCs) that are defined in the architecture design activity.

The CSC is generally made from a set of Computer Software Units (CSUs) that are defined in the architectural and detail design activities. The CSU is often a simple procedure or subroutine.

There are three documents that stand out as important in establishing the development activities and schedule. These documents are: the Software Development Plan (SDP), the Software Requirements Specification (SRS), and the ICD. These three documents generally constitute the definition of a CSCI and are described in subsequent paragraphs.

2.3.1 Software Development Plan

The SDP defines the dates, milestones, and deliverables that will drive the development project. It defines who is responsible for doing what, and by when. It also describes how the important development activities, such as reviews and testing, will be performed. The activities, deliverables, and reviews are described for each step of the development process. The major sections of the SDP are described in the following subsections.

2.3.1.1 Project organization

This section of the SDP lists the performing organizations in the project and describes their responsibilities. It includes how the project will be managed and controlled, the processes and software development practices or standards to be followed by the development team, and the procedures that will be used for tracking and reporting progress.

2.3.1.2 Schedule

The development schedule contains many more tasks than the preliminary timeline presented in the contract Statement of Work (SOW). The SOW enables the project manager to monitor and control progress as the development proceeds. The software developer and the project manager, and often the government customer, develop and agree upon the SDP. The schedule is one of the most critical elements of the SDP and one of the most common factors in project failure. A good estimate is a requirement for project success.

The project plan schedule needs to contain items directly connected to software quality besides the overall project tasks and milestones, such as design tasks, status meetings, functional specification reviews, design reviews, and coding reviews.

2.3.1.3 Software Design Document

The Software Design Document (SDD) describes the complete design of a CSCI. It describes the CSCI as composed of CSCs and Computer Software Units (CSUs).

The SDD describes the allocation of requirements from a CSCI to its CSCs and CSUs. Prior to the PDR, the SDD is entered into the Developmental Configuration for the CSCI. Upon completion of the Physical Configuration Audit (PCA), the SDD, as part of the SPS, is entered into the Product Baseline for the CSCI.

The SDD is used by the developer for three primary purposes, namely:

- Present the preliminary design at the Preliminary Design Review(s).
- Present the detailed design at the Critical Design Review(s).
- Use the design information as the basis for coding each CSU.

The SDD is used by the government to assess the preliminary and detailed design of a CSCI.

2.3.1.4 Quality Plan

The Quality Plan section of the SDP contains an overview of the Quality Assurance and Test Plan, which verifies that the product performs in accordance with the requirements specification and meets all pertinent customer requirements.

The Test Plan is a part of the Quality Plan section of the Software Development Project Plan. It describes the:

- Overall test policy and objectives.
- Responsibility for test case generation.
- Scope of the testing activity: operating systems, computers, features, functions.
- Rules for software acceptance.

A schedule with milestones and the names of people and resources committed to achieving them should be part of this plan.

2.3.2 Software Requirements Specification

The SRS is produced at the conclusion of the requirements analysis task. The SRS is the key document that defines the product to be produced during the full-scale development following the SRR. Software functions and performance are allocated to the CSCI as part of the systems engineering activity. The SRS contains a refinement of the requirements allocation which includes a complete information description, a detailed functional description, a representation of system behavior, performance requirements and design constraints, and software validation criteria.

2.3.3 Interface Control Document

The SRS is accompanied by an interface description in the form of an ICD or an Interface Requirements Specification. The interface description defines, in detail, the interface between the CSCI and the higher-layer system hardware and software.

Damn your estimate! Give me two years and a dozen good engineers, and I'll deliver your system.

Hank Beers, experienced software manager, 1985

Section 3

Levels of Detail in Software Estimates

Software estimates generally fall into one of two distinct categories: system-level and component-level. A system-level estimate is the best approach when few or none of the requirements are allocated to CSCIs. On the other hand, the component-level estimate is used to perform detailed estimates of software components at the CSCI level.

A major factor in determining the estimate type is the program acquisition state or phase. Early in the program, generally before or at Milestone A, a system-level estimate should be employed. Currently, there is a general lack of detailed understanding of the requirements, the system has not yet been decomposed into CSCIs, and only a "ballpark" estimate is required. Programs after Milestone B generally use component estimates. Most programs transition from a system estimate to a component estimate as the architecture and the development environment become more fully defined.

A system-level estimate is normally conducted at a time in the system development process before the software subsystem has been decomposed into CSCIs. At that time, the software size is only loosely defined and the software developer may be unknown. This estimate type is generally used for rough planning and resource allocation purposes. System-level estimates produce development effort estimates only and should not be used to predict a schedule.

The component-level estimate is a detailed or CSCI-level estimate. There are two criteria that must be satisfied in order for a component-level estimate to be valid: The testable project requirements must be defined and allocated to one or more CSCIs (as described previously in Figure 2-1), and the CSCI interfaces must be defined.

The software estimating tools available in the marketplace are designed for component-level estimates and should not be used at the system level. Cost estimates obtained for software elements (defined only at the system level using these tools) will be too high, and the schedule estimates for the system-level elements will be too long.

An accurate (or realistic) component estimate requires knowledge of four controlling factors:

- Effective size (Section 6).
- Basic developer capability (Section 8).
- Environment constraints imposed by the development requirements (Section 9).
- Product characteristics, including complexity (Section 10).

3.1 Estimate foundation factors

Software development cost is largely determined by three factors:

- How big is the development?
- How efficient is the developer?

- What are the development constraints?

Size is the measure in lines of code, function points, object points, or some other units, of how large the development effort will be. Cost and schedule are directly related to the size of the product.

Efficiency is a characteristic of the developer. Developer skill and experience lead to higher productivity, lower costs, and shorter schedules.

Every project has constraints. The development environment provides limitations in terms of development system and product requirements. Product complexity limits the size of the development team which, in turn, controls the schedule. Constraints are largely determined by the product (or application) type. For example, constraints applied to accounting systems are less stringent than space system applications. The customer introduces additional constraints related to reliability, equipment volatility, and security, as well as memory and processing constraints.

Development productivity is determined by a combination of developer efficiency and the constraints placed on the development.

Productivity is dealt with in the system-level and component-level models in different ways. The estimating approaches and their differences are described in the following sections.

3.2 System-level estimating model

The system-level (or first-order) model is the most rudimentary estimating model class and is probably the model that will be used most often, especially in early cost and schedule projections. This model is, simply, a productivity constant multiplied by the software product effective size. The result yields a development effort and cost. The productivity constant is defined as the development organization production capability in terms of arbitrary production units. The production units (size) can be effective source lines of code (ESLOC), function points, object points, use cases, and a host of other units depending on one's experience and estimate requirements. For the purpose of this discussion, we will use effective source lines of code (ESLOC) as the production unit measure and person months per ESLOC as the productivity measure. The first-order estimating model can be summarized as:

$$E_d = C_k S_e \qquad (3\text{-}1)$$

where E_d is the development effort in person-months (PM),

C_k is a productivity factor (PM/ESLOC), and

S_e is the number of effective source lines of code (ESLOC).

The effective size can be defined in several ways which are discussed in Sections 4 and 6 in greater detail. For the purpose of this discussion, we fall back to an effective size definition that has been in use since the late 1970s and produces reasonable effective size values for rough estimates. The simple size value is given by

$$S_e = S_{new} + 0.75S_{\text{mod}} + 0.2S_{reused} \qquad (3\text{-}2)$$

where S_e is the software effective size (ESLOC),
S_{new} is the number of new SLOC being created for the product,
S_{mod} is the number of pre-existing SLOC being modified for the product development, and
S_{reused} is the number of pre-existing, unmodified SLOC used in the product (requires functionality allocation and testing).

The productivity factor is commonly determined by the product type, historic developer capability, or both, and is derived from past development projects. This factor includes all the software related effort needed to produce software beginning with architecture design through (and including) final qualification testing and is divided by the effective size. Equation 3-1 is reasonably simple and is widely used to produce high-level, rough estimates.

By collecting several data points from a specific organization developing a specific product type, an average productivity factor can be computed. This value is superior to the factors tabulated in this section for specific projects by a contractor. Collecting several data points of different sizes will produce a size-sensitive productivity relationship. Table 3-1 shows some typical productivity values for various types of software application types.

Table 3-1: Typical productivity factors (person-months per KESLOC) by size and software type

Software Type	D	10 KESLOC	20 KESLOC	50 KESLOC	100 KESLOC	250 KESLOC
Avionics	8	12.6	14.5	17.4	20.0	24.0
Business	15	2.1	2.4	2.9	3.3	4.0
Command & control	10	6.3	7.2	8.7	10.0	12.0
Embedded	8	9.0	10.4	12.4	14.3	17.2
Internet (public)	12	2.1	2.4	2.9	3.3	4.0
Internet (internal)	15	1.1	1.2	1.5	1.7	2.0
Microcode	4-8	12.6	14.5	17.4	20.0	24.0
Process control	12	4.2	4.8	5.8	6.7	8.0
Real-time	8	12.6	14.5	17.4	20.0	24.0
Scientific systems/ Engineering research	12	2.5	2.9	3.5	4.0	4.8
Shrink wrapped/ Packaged	12-15	2.1	2.4	2.9	3.3	4.0
Systems/ Drivers	10	6.3	7.2	8.7	10.0	12.0
Telecommunication	10	6.3	7.2	8.7	10.0	12.0

Source: Adapted and extended from McConnell, *Software Estimation*, 2006, Putnam and Meyers, *Measures for Excellence*, 1992, Putnam and Meyers, *Industrial Strength Software*, 1997 and Putnam and Meyers, *Five Core Metrics*, 2003

The values contained in Table 3-1 have been developed from a wide range of software developers and environments. The software types in the table may not correlate exactly with the product type you are estimating, so some subjectivity is required in selecting the appropriate type to use for your estimate. The productivity factor given for each type is the mean of the data collected for that software type. Also, consider that when the types were defined there was a wide dispersion of productivity values for each size and type. Some data is conservative, but some is quite aggressive; in other

words, data is often collected in a way to increase the apparent productivity factor. For example, some costs for development activities may have been reallocated from development to test or to system engineering. The mean value is taken from a skewed distribution that, in most cases, will make the estimate somewhat optimistic. Be aware of the data!

The system complexity (D) column in Table 3-1 shows a relationship between the application type and the typical software complexity. Complexity is defined as the degree to which systems design or implementation is difficult to understand and verify. System complexity values have a typical range, in practice, between 4 and 15, with 4 being the most complex. A comprehensive discussion of software complexity is contained in Section 10.1. It is interesting to note the productivity rate achieved for each software type tends to group around the associated complexity value.

Issues arise when the effective project size is less than approximately 20 KESLOC. First, the data from which the tables are generated have a wide variance; that is, the productivity values for those projects are highly dependent on individual performance. The high variation adds risk to the tabulated values. This is explained in more detail in Section 4.

Second, when tasks are 20 KESLOC or less, it is highly likely that the software task will be developed as a component (CSCI). At the component level, environment and personnel effects will be factored into the productivity value. The component-level estimating approach is discussed in Section 5.

The ideal CSCI size is reputed to be about 25-30 KESLOC, resulting in a development team of approximately 15 people, in a normal environment. This is because the ability of the development personnel to communicate and work together is important in small projects.

The 100 KESLOC productivity rate listed in Table 3-1 is close to the rate expected for medium-size projects but the size is on the high side for a single-CSCI project.

Columns 3 and 4 of the table are highlighted to emphasize the increased productivity normally present in small development teams. The data in Table 3-1 for project sizes of 50 KESLOC or greater is more stable. It is recommended that values below the 50 KESLOC column of the productivity factor table be used with care because of the large variance in the underlying data.

It stands to reason that military software development programs result in lower productivities. The high levels of reliability and security inherent in military applications necessitate more careful analysis and design as well as more exhaustive testing than the average commercial or web application.

There are *no* useful methods to project a development schedule at the system level unless the system can be developed as a single CSCI.

3.3 Component-level estimating model

The component-level estimate requires an allocation of the system requirements into CSCIs. The CSCIs can be further allocated into subunits to achieve proper sizing for development. Some components may be reused from other previously developed subunits or include the use of a COTS product.

The component-level estimate should not be used for software development task estimates with "pieces" larger than 200 KESLOC. This is due to the impracticality of a development team working together effectively, given the team size necessary to implement a CSCI of this magnitude.

The 250 KESLOC project is beyond the upper limit for a single CSCI project. The Aerospace Corporation report on DoD software project development[12] shows that no CSCIs larger than 200 KESLOC in their database have ever been successfully completed, as illustrated in Figure 3-1. The team for a project of this size approaches 100 people working and communicating as a team. As a project, the 250 KESLOC total size needs to be decomposed into a group of CSCIs that can be independently managed and developed.

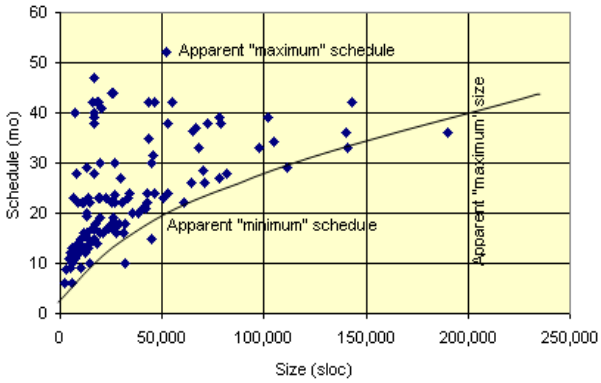


Figure 3-1: Achievable software development effective size and schedule

The component-level estimate addresses some of the weakness of the system-level estimate. The first system-level weakness is its inability to adjust the productivity factor to account for variations between projects and differences in development environments. For example, contractor A may have a more efficient process than contractor B; however, contractor A may be using a development team with less experience than assumed in the historic productivity factor. Different constraints may be present in the current development than were present in previous projects. In addition, using a fixed (or calibrated) productivity factor limits the model's application across a wide variety of environments.

A second major weakness is the inability to accurately plan a development schedule for the project. The component-level estimate, unlike the system-level estimate, can focus on the development team and environment to produce a schedule estimate and staffing profile, as well as a development effort estimate.

The component-level estimating model compensates for the system model's narrow applicability by incorporating a set of environment factors to adjust the productivity factor to account for a wider range of variables such as team capability and experience, requirements volatility, and product constraints. The form of the component-level estimating model is

[12] Long, L., K. Bell, J. Gayek, and R. Larson. "Software Cost and Productivity Model." Aerospace Report No. ATR-2004(8311)-1. Aerospace Corporation. El Segundo, CA: 20 Feb 2004.

$$E_d = C_k \left(\prod_{i=1}^{n} f_i \right) S_e^{\beta} \qquad (3\text{-}3)$$

where E_d is the full-scale development effort (SRR through FQT),
f_i is the ith environment factor,
n is the number of environment factors, and
β is an "entropy[13]" factor that measures the communication efficiency of the development team.

The number of environment factors varies across estimating models, and is typically between 15 (COCOMO) and 32 (Jensen). Models using lower numbers of factors ignore management impacts and take a higher-level view of other environment factor impacts on productivity.

Environment factors can be grouped into four distinct types: personnel, management, environment, and product. A table of common environment factors is shown in Appendix I-11.

The component model compensates for the productivity decrease in larger projects by incorporating an entropy factor (β) to account for the change. The entropy effect demonstrates the impact of the large number of communications paths that are present in large development teams. The development team theoretically has $n(n-1)/2$ communication paths, where n is the number of development personnel. An entropy factor value of 1.0 represents no productivity change with size. An entropy value of less than 1.0 shows a productivity increase with size, and a value greater than 1.0 represents a productivity decrease with size. Entropy values of less than 1.0 are inconsistent with historical software data[14]. Most of the widely used models in the 1980s (COCOMO embedded mode, PRICE®-S, Revised Intermediate COCOMO, Seer and SLIM®) used entropy values of approximately 1.2 for DoD projects. The 1.2 value applies to development tasks using more than five development personnel. A lone programmer task has a corresponding entropy factor of 1.0 since there is no communication issue.

If the system can be built as a single CSCI, the schedule can be approximated by an equation of the form

[13] Entropy is defined as "An index of the degree in which the total energy of a thermodynamic system is uniformly distributed and is thus unavailable for conversion into work." Entropy in software development equates to the energy expended in the process which does not contribute to the software product. For example, reverse engineering of the existing software product is necessary, but produces no product. Communication faults create errors, rework, and consumes resources, but does not contribute to the product.

[14] This is a good place to point out a major difference between software and almost any other manufactured product. In other estimating disciplines, a large number of product replications improve productivity through an ability to spread costs over a large sample and reduce learning curve effects. The software product is but a single production item which becomes more complex to manage and develop as effective size increases.

$$T_d = X * \sqrt[3]{E_d} \text{ months} \qquad (3\text{-}4)$$

where T_d is the development time (months),

X is a constant between 2.9 and 3.6, depending on the software complexity, and

E_d is the development effort (PM).

3.4 Estimating process

To summarize the software estimating approaches, there are two fundamental estimating processes for software development: system level, and component level. Common to both processes are the effective size and complexity of the system under development

The system process is generally used in the early phases of development when details, such as an architectural design, are not available. The system may be only a concept at this stage or may have functional requirements, but in any case, the system does not have the components defined at this point. The information available probably includes a size estimate and an application type. From the application type we can derive an approximate value for the system complexity. The estimate developed under these conditions is only a coarse approximation. Adding significant digits to the results does not improve accuracy or reality.

The validity of an input data value is inversely proportional to the number of trailing zeros in its representation.

Figure 3-2 shows the relationship between system-level and component-level estimates. The system-level estimating process, based on size and productivity information, is described in Section 4.

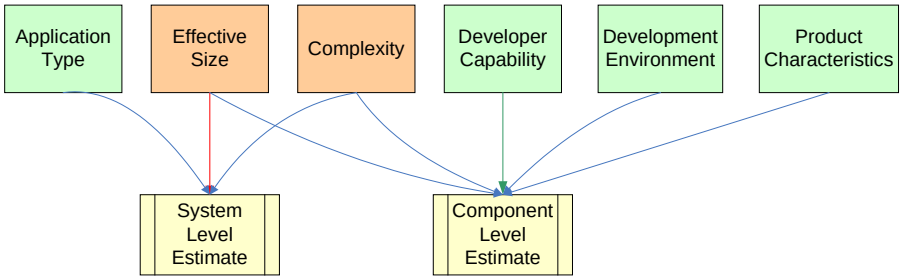


Figure 3-2: Software elements and their relationship to the estimate types

The component-level process is generally used after the software components are defined with an SRS to describe the component functionality and an ICD to define the component interface to the larger system. These documents describe the scope of the components and make it possible to estimate the effort, schedule and staffing characteristics of the actual development. At this point, the developer (or a generic developer), with the capabilities to construct the software component has been defined. The details of the component-level estimating process are described in Section 5.

We learn from experience that we don't learn from experience.

D.H. Lawrence

Section 4

System-Level Estimating Process

The system-level estimating process is generally used in the early phases of development, where details such as an architectural design are not available. The system may only be a concept or may have functional requirements, but in any case, does not have the components defined. The information available probably includes a size estimate and an application type, as shown in Table 4-1. From the application type, we can derive an approximate value for the system complexity. The estimate developed under these conditions is only a coarse approximation. Adding significant digits to the results does not improve accuracy or reflect reality. Also, note that the number of trailing zeros is a likely indication of the uncertainty of the number.

Table 4-1: System Concept Information

Application Type	Baseline Size
System Manager	219,000
Control Data Links	320,000
Sensors	157,000
Operating System (Application Program Interfaces)	53,000
System Total	749,000

The system-level estimating model is defined as:

$$E_d = C_k S_e \qquad (4\text{-}1)$$

where E_d is the development effort in Person-Months (PM),

C_k is a productivity factor that depends upon the product type and the development environment (PM/ESLOC), and

S_e is the number of ESLOC (size).

The development effort can be in Person-Hours, PH, or any other unit corresponding to a measure of productivity.

If – and only if – the system can be built as a single CSCI, the schedule can be approximated by an equation of the form:

$$T_d = X * \sqrt[3]{E_d} \text{ months} \qquad (4\text{-}2)$$

where T_d is the development time (months),

X is a constant between 2.9 and 3.6 depending upon the software complexity, and

E_d is the development effort (PM).

> *A CSCI with an effective size greater than 200,000 source lines will never be completed successfully. Also, if the required development schedule exceeds five years, the CSCI will never be delivered.*
>
> Results of Aerospace Corp data analysis

For example, if a software subsystem is projected to be 300,000 ESLOC and the associated productivity factor is 100 ESLOC per person month, the development effort estimate will be 3,000 PM. The corresponding development time, calculated using most software estimating tools, is approximately 50 months and 60 developers, or

60 developers = 3,000 PM / 50 MO.

If the subsystem is developed as five equal CSCIs, the development time is approximately 29 months plus some subsystem integration time. In any case, the estimated development times for the two options diverge greatly and demonstrate the weakness of the development schedule estimate at the system level. Even without using estimating tools, it is logical that the required schedule for the two options will be widely disparate.

Equation (4-1) states that two important pieces of information – productivity and size – are necessary to conduct a software cost analysis at the system

level. There is a straightforward procedure performing a complete system-level cost analysis. Briefly, the steps are:

1. Construct a list of the software elements contained in the system. These elements can be major subsystems, CSCIs, or both, depending upon the descriptive information available. Most often, a system-level estimate is performed prior to, or slightly after, Milestone A.
2. Determine the complexity of each system software elements (Section 4.1). As an alternative approach, use the productivity factor table (Table 4-4 in Section 4.4) which contains typical complexity values for each of the 13 application categories.
3. Determine the effective size for each element in the system software list. The element size information may be available in many forms including ESLOC, function points, use cases, object points, or other measures. The important point is that the form must ultimately be reducible to ESLOC to be usable by most estimating tools.
    a. Determine the baseline size estimate for each element (Section 4.2.1 for source code estimates, Section 4.2.2 for function points).
    b. Calculate the projected size growth for each element taking into account the maturity of the project and the element complexity (Section 4.3).
4. Choose a productivity factor for each software element (Section 4.4).
5. Calculate the development cost for each element (Section 4.5).
6. Validate the development cost estimate (Section 4.6).
7. Allocate the development costs to the project phases (Section 4.7).

The system-level cost estimating process described in this section is supported by a case study that demonstrates development of an estimate from inception through completion. This example is contained in Appendix F.

4.1 Product complexity

A list of the system's software elements may be available from the ICD or other design and architecture documents prior to Milestone A. These documents may also contain other important estimating information such as planned size and complexity of the software elements.

Complexity (apparent) is defined as the degree to which a system or component has a design or implementation that is difficult to understand and verify[15]. The complexity number may be viewed as analogous to the amount of code the person can understand or maintain, thus the lower a number the more intricate or complex. In more visible terms, complexity, or "*D*" (for difficulty), is a function of the internal logic, the number and intricacy of the interfaces, and the understandability of the architecture and code. The first thing to understand about complexity is that it has nothing to do with software size.

[15] Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY: 1990.

Larry Putnam[16] empirically noted that when his software database was plotted as K (total lifecycle effort in person years) vs. T^3 (development time in years), the data stratified according to the complexity of the software system.

Stratification of the complexity data occurs around the system types. Table 4-2 is a useful rough guide for determining the appropriate D value for specific software types. The majority of estimating tools refer to D as complexity.

Table 4-2: Stratification of complexity data

D	Description
4	Development primarily using microcode. Signal processing systems with extremely complex interfaces and control logic.
8	New systems with significant interface and interaction requirements with larger system structure. Operating systems and real-time processing with significant logical code.
12	Application with significant logical complexity. Some changes in the operating system, but little or no real-time processing.
15	New standalone systems developed on firm operating systems. Minimal interface with underlying operating system or other system parts.
21	Software with low logical complexity using straightforward I/O and primarily internal data storage.
28	Extremely simple software containing primarily straight-line code using only internal arrays for data storage.

4.2 Size estimating process

Size is the most important effort (cost) and schedule driver in the software development estimate. Size information early in a project is seldom, if ever, accurate. Information presented in a Cost Analysis Requirements Description (CARD) is generally predicted during the concept development phase, prior to the start of full-scale development. Size information is often presented as a single value leading to a point cost and schedule estimate; however, the estimate is more accurately expressed as a minimum (most optimistic) value, a most likely value, and a maximum (most pessimistic) value to give a more realistic picture of the information.

The size values used in a software cost and schedule estimate are derived from a three-step process:

1. Determine the baseline effective size value and the baseline size standard deviation. If the system contains function points as part of the size specification, the function points must be converted into equivalent ESLOC as part of this step.
2. Determine the maximum effective size growth. The maximum growth value is determined from the developer capability, the maturity of the software product, the product complexity, and the anticipated reuse level of pre-existing source code.
3. Determine the mean growth estimate. This is used as the most likely size value in most cost and schedule estimates.

4.2.1 Effective source lines of code (ESLOC)

A SLOC is a measure of software size. A simple, concise definition of a source line of code is: any software statement that must be ***designed***, ***documented***, and ***tested***. The three criteria designed, documented, and tested, must be satisfied in each counted SLOC.

ESLOC, as a measure of work, is a useful and necessary concept for defining "effective size" for effort (cost) calculations. Development ESLOC are numerically identical to physical source lines when the total software product is being developed from scratch or as new source lines. Unfortunately, most software developments incorporate a combination of new source code, existing code modified to meet new requirements, and unmodified code or reusable software components. A detailed description of the concept of work and effective size is contained in Section 6.

[16] Putnam, L.H. "A General Empirical Solution to the Macro Software Sizing and Estimating Problem." IEEE Transactions on Software Engineering. New York, NY: 1978.

4.2.2 Function point counting

Function point analysis is a method for predicting the total size (and ONLY the total size – not the development effort or time) of a software system. Function points measure software size by quantifying the system functionality provided to the estimator and is based primarily on the system logical design. It is difficult to use function points as a measure of size in modified systems except for new and isolatable functionality. (Functionality cannot be changed without modifying source code, but source code *can* be modified *without* affecting functionality (the "zero function point" problem.).

The overall objective of the function point sizing process is to determine an adjusted function point (AFP) count that represents the functional size of the software system. There are several steps necessary to accomplish this task. They are:

1. Determine the application boundary.
2. Identify and rate transactional function types to determine their contribution to the unadjusted function point (UFP) count.
3. Identify and rate data function types to determine their contribution to the unadjusted function point count.
4. Determine the value adjustment factor (VAF) from the 14 General System Characteristics (GSCs).
5. Multiply the UFP count by VAF to obtain the adjusted function point count (AFP).

The function point counting process is explained in more detail and demonstrated in Section 6.7.

4.3 Software size growth

Some experts consider software code growth (Figure 4-1) to be the single most important factor in development cost and schedule overruns. Code growth is caused by a number of factors: requirements volatility, size projection errors, product functionality changes, and human errors.

The maximum size growth factor is shown in Table 4-3. The growth factor is determined by the complexity of the software product and from the maturity of the product at the point the estimate is being made. The maturity, or *M*, value is measured on a 100-point scale where the concept-level maturity is zero and the end of development maturity (or Formal Qualification Test [FQT]) is 100 points.

The maturity factor for each milestone is shown in the table. The factors project the maximum product size at the end of development. The highlighted row in Table 4-3 corresponds to the relative project maturity (M) of 52 at the start of Software Requirements Review (SRR).

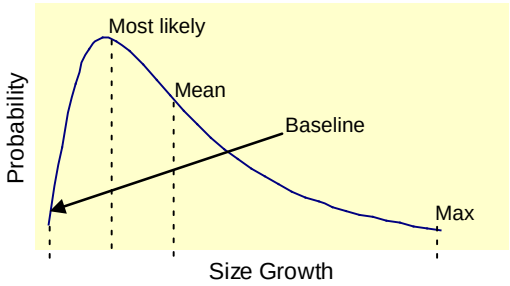


Figure 4-1: Effective size growth distribution

Table 4-3: **Maximum software growth** projections as a function of project maturity and product complexity.

		Complexity							
Maturity	**M**	**≤ 8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Concept	0	2.70	2.53	2.36	2.19	2.01	1.84	1.67	1.50
Source	15	2.47	2.32	2.17	2.03	1.88	1.73	1.59	1.44
C/A	33	2.19	2.07	1.95	1.84	1.72	1.60	1.49	1.37
SRR	**52**	**1.89**	**1.81**	**1.72**	**1.64**	**1.55**	**1.46**	**1.38**	**1.29**
PDR	67	1.66	1.60	1.54	1.48	1.42	1.35	1.29	1.23
CDR	84	1.40	1.36	1.33	1.30	1.26	1.23	1.20	1.16
FQT	100	1.15	1.14	1.14	1.13	1.12	1.11	1.11	1.10

Table 4-4 shows the mean projected size growth as a function of project maturity and product complexity that corresponds to the maximum growth illustrated in Figure 4-1.

Size growth data for complexity values less than eight is too limited to be of any value; hence, the information in Tables 4-3 and 4-4 present growth information for complexities in the range of eight to 15. For components of higher complexity ($D = 4$), assume the growth value for $D = 8$.

Table 4-4: **Mean software growth** projections as a function of project maturity and product complexity.

		Complexity							
Maturity	**M**	**≤ 8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Concept	0	1.50	1.45	1.40	1.35	1.30	1.25	1.20	1.15
Source	15	1.43	1.39	1.34	1.30	1.26	1.22	1.17	1.13
C/A	33	1.35	1.31	1.28	1.24	1.21	1.18	1.14	1.11
SRR	**52**	**1.26**	**1.24**	**1.21**	**1.19**	**1.16**	**1.13**	**1.11**	**1.09**
PDR	67	1.19	1.18	1.16	1.14	1.12	1.10	1.09	1.07
CDR	84	1.12	1.11	1.10	1.09	1.08	1.07	1.06	1.05
FQT	100	1.04	1.04	1.04	1.04	1.04	1.03	1.03	1.03

The cost and schedule projections are normally computed from the mean size growth value. This is the value that applies most for a realistic estimate. The probability of the baseline size existing at the end of development is very small and the corresponding delivery schedule and development effort are unlikely (see Section 6.5 regarding size growth).

4.4 Productivity factor

There are two methods described in this guidebook to determine the productivity factor for software development. Both yield reasonable –albeit different – values. Remember, the results produced here are estimates. Without historic data that describes the characteristics of the product, the software development environment, and the developer capability, it is unlikely the productivity factors, here or anywhere else, are going to produce identical effort and schedule estimates. At the system level, the product characteristics are likely to be known, but the developer is not typically identified at the time the system estimate is made.

4.4.1 Productivity factor table

A table of system types, shown in Table 4-5, is divided into 13 categories to provide meaningful productivity factors for these types of systems. Additional system types can be covered by selecting the given type using the closest approximation. Table 4-5 assumes an average development organization operating in a normal environment. Subjective adjustments can be made to the factors to compensate for application experience. The complexity value *D* represents the value that is typical for that software type. Complexity is discussed in detail in Section 10.1.

Table 4-5: Typical productivity factors (ESLOC per person month) by size and software type

Software Type	D	10 KESLOC	20 KESLOC	50 KESLOC	100 KESLOC	250 KESLOC
Avionics	8	79	69	57	50	42
Business	15	475	414	345	300	250
Command & control	10	158	138	115	100	83
Embedded	8	111	97	80	70	58
Internet (public)	12	475	414	345	300	250
Internet (internal)	15	951	828	689	600	500
Microcode	4-8	79	69	57	50	42
Process control	12	238	207	172	150	125
Real-time	8	79	69	57	50	42
Scientific systems/ Engineering research	12	396	345	287	250	208
Shrink wrapped/ Packaged	12-15	475	414	345	300	250
Systems/ Drivers	10	158	138	115	100	83
Telecommunication	10	158	138	115	100	83

Source: Adapted and extended from McConnell, *Software Estimation*, 2006, Putnam and Meyers, *Measures for Excellence*, 1992, Putnam and Meyers, *Industrial Strength Software*, 1997 and Putnam and Meyers, *Five Core Metrics*, 2003.

The utility of the system model can be effective early in the development process before the software system architecture is established and the software components are defined. The model is also valuable where there is little or no knowledge of the developer or the developer's environment available. A reasonable "ballpark" estimate can be created quickly to support rough planning.

4.4.2 ESC metrics

The Air Force Electronic Systems Center (ESC) compiled a database[17] of military projects developed during the years 1970-1993. The ESC categorization follows the reliability definitions posed by Boehm in Software

[17] AFMC Electronic Systems Center. Cost Analysis: Software Factors and Estimating Relationships. ESCP 173-2B. Hanscom AFB, MA: 1994.

Engineering Economics; that is, grouping reliability according to the following categories[18] shown in Table 4-6.

Table 4-6: ESC reliability categories

Category	Definition
Very low	The effect of a software failure is the inconvenience incumbent on the developers to fix the fault.
Low	The effect of a software failure is a low level, easily-recoverable loss to users.
Nominal	Software failure causes a moderate loss to users, but a situation recoverable without extreme penalty.
High	The effect of a software failure can be a major financial loss or a massive human inconvenience.
Very high	The effect of a software failure can be the loss of human life.

Each project contains CSCIs with different ratings. Each CSCI within a project is required to interface with one or more additional CSCIs as part of a composite system. The interaction includes both internal interfaces as well as interfaces to external systems. The number of integrating CSCIs is defined as the total number of CSCIs in the project. ESC formed three categories for their productivity analysis based on the number of integrating CSCIs and the required reliability level, as shown in Table 4-7.

Table 4-7: Definition of complexity/reliability categories

	Integrating CSCIs		
Reliability	**0-6 CSCIs**	**7-10 CSCIs**	**> 10 CSCIs**
Very low - Nominal (Moderate loss)	Category 1	Category 1	No Data
High (Major Financial Loss)	Category 2	Category 2	Category 3
Very high (Public safety required)	Category 2	Category 3	Category 3

4.5 System-level cost estimating

The category drawn from Table 4-7 specifies the productivity factor in Table 4-8 to be used for the system level estimate. An interesting note to the ESC productivity analysis shown in Table 4-8 is the narrow range of the productivity data. Standard deviations for the three categories even without considering the project size and development environment are very low for the 93 projects contained in the database. Note that there is no standard for defining productivity. The standard in the guidebook is ESLOC/PM. However, ESC uses PM/KSLOC, so it is proper to use their definition in their example.

[18] Boehm. B.W. Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall, 1981.

Table 4-8: Productivities for military applications by category

Project Type	Productivity (SLOC/PM)	Productivity Factor (PM/KSLOC)	Productivity Range (SLOC/PM)	Standard Deviation (SLOC/PM)
All Programs	131.7	7.60	n/a	n/a
Category 1	195.7	5.10	116.9 – 260.8	49
Category 2	123.8	8.08	88 – 165.6	23.6
Category 3	69.1	14.47	40.6 – 95.2	16.5

System-level cost estimates, using either the productivity table approach extended from the McConnell-Putnam category definitions or from the ESC model, can quickly produce credible "ballpark" estimates. Note that *quickly* does not equate to *simply*.

The ESC model is derived from a collection of military projects. The productivity table model is derived from a large set of projects containing both military and commercial developments. The ESC model is also size-independent. Most of the projects investigated by ESC are large and not categorized by size. Reliability is not considered in the productivity table model except where common to each of the 13 system categories.

An estimate using the baseline effective software size is the most optimistic cost estimate since it assumes there will be no software growth during full-scale development of the product. The baseline estimate value is useful only if the lowest possible cost is of interest. An estimate using the mean potential growth value is usually considered the most likely cost estimate. The maximum effective size growth value is the estimate that has the most meaning in a cost and/or schedule risk analysis. The ESC model does not reflect the productivity decrease that occurs as the effective size increases with growth.

4.6 Reality check

All estimates should be validated. Most of the historic data we look at is based on the cost and schedule for the development of CSCIs. The productivity table information in Table 4-5 is a good source of historical validation data for a wide range of systems. Judgment was used in defining the 13 product types and sorting the product development information into those types.

A second part of any reasonable estimate is the consideration of risk. Software risk often presents itself as an increase in effective size. Large subsystems can be decomposed into smaller CSCIs which will, in turn, decrease the required schedule by developing smaller components. The smaller components will achieve higher productivity which will reduce development cost.

4.7 Allocate development effort

The distribution of effort varies with the effective size of the system. The larger the system, the more effort will be spent in system test and integration. The relative effort spent in creating the architecture increases with size as well. The rough effort distribution as a function of source size is presented in Table 4-9.

Table 4-9: Total project effort distribution as a function of product size

Size (KSLOC)	Activity				
	Rqmts. (%)	High-Level Design (%)	Develop (%)	Sys. Test (%)	Mgt. (%)
1	4	12	77	17	11
25	4	20	67	24	13
125	7	22	63	25	15
500	8	25	56	32	19

Adapted from McConnell, 2006; Putnam and Myers, 1992; Jones, 2000; Boehm et al, 2000; Putnam and Myers, 2003; Boehm and Turner, 2004; Stutzke, 2005

Each row of the table adds up to a percentage greater than 100% since the requirements effort and system test effort are not included in the full scale development effort resulting from Equation (4-1). Full scale development includes the high level design (architecture), the development (detail design, code and unit test), internal component integration, and the technical management supporting the development activities. The requirements definition effort must be added on to the beginning of the development and system integration to the end. Software maintenance is a separate calculation.

The next step in the effort estimate is to calculate the effort to be assigned to each of the individual development and integration activities. If system integration is to be included in the total system development effort, the development computed in the last step must be increased to account for this. The total effort is computed from the relationship:

$$E_{Total} = k \times E_{Develop} \qquad (4\text{-}3)$$

where E_{Total} is the effort including both full-scale development and the CSCI integration effort,

$E_{Develop}$ is the full-scale development effort, and

k = 1.21 to 1.40 (an additional 21 to 40 percent effort) depending on the anticipated system integration difficulty.

A commercial standalone software product development may entail no system integration. However, a typical integration factor k is in the range of 1.21 to 1.40 for normal system integration requirements. An integration factor value of 1.28 is a reasonable default for CSCI-system integration.

Note that the effort expenditures in system development are not constant across all activities. Some of the variance is due to different schedule periods for each activity. Most of the variance, however, is due to a peculiarity in the way successful software development projects are staffed from the start of development through delivery. The staffing profile that correlates well with successful projects will be described in Section 5.1. It is known as the Rayleigh-Norden profile (named for the men who discovered the presence of the phenomenon) and projects staff at a linear rate at the start that is inversely proportional to the project complexity. The complexity aspect is not visible at the system level, but becomes an important concept at the component level of the project.

Remember, the system-level estimating model does not allow for the prediction, or projection, of a software development schedule. The schedule

can only be produced when the system has been decomposed to the component (CSCI) level and estimated using a component level estimating technique.

4.8 Allocate maintenance effort

Life-cycle cost has multiple meanings. A common definition relates to the software development process where the life cycle includes all development phases from requirements analysis through system integration. A more realistic definition includes all phases from requirements analysis to disposal at the end of the software's life. The latter definition includes the maintenance of the software product, which can be many times more than the cost of development. It is common for the cumulative cost of maintenance to exceed the cost of full-scale development by the end of the fifth calendar year of operations.

Software maintenance costs encompass two major categories: enhancing the software, and retaining knowledge of the software product. Enhancements are of two types: (1) changing the product's functional specification (adding new capability), and (2) improving the product without changing the functional specification. Enhancement is the process of modifying the software to fit the user's needs. The activities we normally relate to maintenance, which involve making changes to the software, all fall into the enhancement category.

Software product improvement falls into three subcategories[19]: corrective, adaptive, and perfective.

- Corrective maintenance: processing, performance, or implementation failures
- Adaptive maintenance: changes in the processing or data environment
- Perfective maintenance: enhancing performance or maintainability

The second maintenance category, knowledge retention, is often ignored in the estimate. The number of people charged with the maintenance task must be large enough to retain knowledge of the inner workings of the software product; that is, a staff must be retained who have detailed knowledge of the software. For large software products, the number is typically not small.

The system-level software estimating model does not have adequate information to project the defect removal effort. There is no information describing the development that is essential for the calculation. However, there is adequate information to deal with the enhancement and knowledge retention costs.

4.8.1 Software enhancement

The enhancement effort uses the concept of Annual Change Traffic (ACT), which is the decimal fraction of the total product source code that will undergo change or modification during a year and includes all types of enhancements.

The enhancement effort component is given by the expression:

[19] Swanson, E.B. The Dimensions of Maintenance. Proc. of the IEEE/ACM Second International Conference on Software Engineering. Oct. 1976.

$$E_1 = \frac{ACT}{gPF} S_{total} \text{ (PM/YR)} \quad (4\text{-}4)$$

where E_1 is the enhancement effort (PM per year)

ACT is the annual change traffic (decimal fraction),

g is the relative quality of the maintenance organization (0.5 to 1.7),

PF is the software development productivity factor (ESLOC/PM), and

S_{total} is the total software size (SLOC).

As a rule of thumb, based upon historical data, once the enhancement effort is greater than 20 or 30 percent, it is more practical to consider starting a fresh development to incorporate the changes or enhancements into the system. Imagine that the ACT value was 1.0. That implies the system would be completely replaced within one year of maintenance. It is not very likely that this would happen, especially if the initial system development took three or four years.

The productivity of the maintenance organization is determined relative to the quality (efficiency) of the development organization. If the development organization is maintaining the software, the value g is 1.0. Often the maintenance organization is not as experienced or competent as the development team. In that case, the g factor must be decreased (e.g., 0.8).

4.8.2 Knowledge retention

Operational knowledge of the software is essential to maintaining the product over its lifetime (i.e., changing and/or enhancing the software product). The knowledge retention effort is often ignored in forecasting the effort required to keep the product functioning. Ignoring the operation knowledge retention effort leads to either severe cost overruns or poor product knowledge during the maintenance activity.

The number of personnel required to retain operational knowledge of the software product can be projected by utilizing a concept that persists from early software folklore. An old heuristic "Each programmer can maintain four boxes of cards" (where a card box contained 2,000 computer punch cards) is almost as old as software itself. An updated version of the heuristic that is very applicable today is:

$$E_2 = \frac{9.2}{gD} S_{total} \text{ PM per year} \quad (4\text{-}5)$$

where E_2 is the enhancement effort (PM per year),

D is the effective software product complexity,

g is the relative quality of the maintenance organization (0.5 to 1.5) with 1.5 matched with exceptional organizations, and

S_{total} is the total software size (KSLOC).

4.8.3 Steady state maintenance effort

The steady-state maintenance effort per year is given by:

$$E_m = \max(E_1, E_2) \text{ (PM/YR)} \qquad (4\text{-}6)$$

where E_m is the steady-state software product maintenance effort.

Equation (4-6) shows that the personnel performing enhancement of the software product are also those holding the operational knowledge of the product. Only in the case where the enhancement requirements exceed the number of personnel necessary to retain operational knowledge of the product are additional personnel added to the maintenance task. Often, the operational knowledge requirements are much larger than the enhancement requirements. Then, maintenance effort can be computed from Equation (4-5) alone.

The devil is in the details.
Anonymous

Section 5

Component-Level Estimating Process

The component-level estimating process is generally used after details such as an architectural design for the system are available. Programs after Milestone B generally support component level estimates. The system definition will have functional component requirements defined, namely Software Requirements Specifications (SRSs) and the interfaces between components defined by Interface Control Documents (ICDs) or Interface Requirements Specifications. A component is a single CSCI. The information available also includes a size estimate and an application type as shown in Table 5-1. The example component sizes and functionality were derived from an actual system with the names changed.

Table 5-1: CSCI Size Estimates

Task Name	Size, Eff. Baseline (ESLOC)
Control Data Links	236,000
Satellite	152,000
Transmitter	96,000
Receiver	56,000
Radio	84,000
Brain	184,000
Sensors	157,000
OS (APIs)	53,000
System Total	630,000

The developer may also be known at this time, or at least the developer class will be known well enough that the typical characteristics can be defined.

The component-level estimate model compensates for the system model's narrow applicability by incorporating a description of the developer capability, a set of environment factors that adjust the productivity factor to fit a wider range of problems, and a set of product characteristics. The development effort is equal to the product of a proportionality constant, a set of adjustment factors that account for the productivity impacts of the development environment and the product constraints, as well as the effective size adjusted for communication issues. The generalized mathematical form of this model is:

$$E_d = C_k \left(\prod_{i=1}^{n} f_i \right) S_e^{\beta} \qquad (5\text{-}1)$$

where

C_k = proportionality constant

f_i = the ith environment factor,

n = the number of environment factors,

S_e = effective software size, and

β = an entropy factor that measures the communication efficiency in the development team.

Caution: The component-level tools should not be applied to system-level estimates. The underlying architecture of the model is not compatible with the assumption at the heart of system-level estimates.

The environment factors can be grouped into four distinct categories: personnel, management, environment, and product. Table I-11 shows the common environment factors.

The component model compensates for the productivity decrease in large projects by incorporating an "entropy" factor to account for the productivity change with effective size. The entropy factor accounts for the impact of the large number of communications paths that are present in large development teams. The development team theoretically has $n(n-1)/2$ communication paths where n is the number of development personnel. The entropy factor is similar to the productivity variation with effective size shown in Table 4-5.

While the component estimating model requires considerably more information from the estimator than is required by the system-level model, the extra effort is compensated for by the additional capabilities of the tools that implement the various forms of the component model. Manual calculation using the component model is more tedious, but offers greater accuracy and reward for the effort. For example, the system-level approach in Section 4 cannot project a development schedule. Most component-level models do provide that capability.

The number of environment parameters varies from 15 to 32, depending upon the particular implementation of the model as described in Equation (5-1). The 1981 implementation of COCOMO I described by Boehm[20] uses only 16 parameters. COCOMO II[21] requires 23 parameters. Implementations of the Jensen model,[22] including Sage and SEER-SEM™, require 32 parameters.

Most of the tedium involved in component model use can be bypassed by using a set of default environment parameter values. However, the accuracy of the component model is decreased significantly if the default value set is used because of assumptions made in the default environment. Many estimating tools offer parameter sets or templates that describe many environment and product-type options. It is important to understand how the parameter set options are combined for a specific estimate.

In the component-level model, a development schedule for each component can be approximated by an equation of the form:

$$T_d = X * \sqrt[3]{E_d} \text{ months} \qquad (5\text{-}2)$$

where T_d = development time (months),

X = an empirically derived constant between 2.9 and 3.6 depending on software product complexity, and

E_d = development effort (PM).

Equation (5-2) gives the optimum staffing profile's minimum schedule. At this point, we will describe a straightforward procedure for a system-level cost analysis. The steps are, briefly:

1. Construct a list of the software elements (CSCIs) contained in the software system. These elements <u>must</u> be CSCIs, which limits the initiation of a component-level estimate until after the CSCIs are defined. This condition is generally satisfied after Milestone B.
2. Determine the complexity of each CSCI element (Section 5.2). Note: Some component-level model implementations consider complexity to be a product descriptor in the environment.
3. Determine the effective size of each element in the system software list. The element size information may be available in many forms including

In most estimating tools, the input software size form is converted to SLOC internally before computing the development cost and schedule.

[20] Boehm, B.W. <u>Software Engineering Economics</u>. Prentice-Hall. Englewood Cliffs, NJ: 1981.
[21] Boehm, B.W., A. Egyed, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby. "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0." <u>Annals of Software Engineering</u>. 1995: 295-321.
[22] Jensen, R.W. <u>A Macrolevel Software Development Cost Estimation Methodology</u>. Proc. of the Fourteenth Asilomar Conference on Circuits, Systems, and Computers. Pacific Grove, CA., Nov.17-19, 1980.

source lines of code, function points, use cases, object points, or other measures.

   a. Determine the baseline size estimate for each element (Section 5.3.1, and 6.4 for source code estimates, Sections 5.3.2 and 6.7 for function points).

   b. Calculate the projected size growth for each element depending upon the maturity of the project and the element complexity (Section 6.5).

4. Determine the impact of the development environment (effective technology constant) from the set of environment factors. The effective technology constant serves as the productivity factor for each CSCI (Sections 8 through 10).

5. Calculate the development effort for each CSCI in the system (Section 5.6).

6. Calculate the development schedule for each CSCI in the system.

7. Allocate the development effort and schedule to the project phases. (Section 5.8).

8. Validate the development effort and schedule estimates (Section 5.7). Compare the productivity obtained for the estimates with the historical development productivity for similar products. The productivity results should be similar.

Size growth is calculated automatically in the Sage model.

Process steps 3 through 7 are generally calculated, or supported, in the commercial software estimating tools.

The component-level software cost and schedule estimating process described in this section is supported by a component-level case study in Appendix G that demonstrates the development of an estimate from inception through completion. Evolving fragments of the estimate are updated as each step or portion of a step is completed.

5.1 Staffing profiles

The Peter Norden study[23] at IBM showed the maximum staffing rates for successful research and development projects did not exceed the maximum staffing rate defined for the associated project type. Norden asserted that an engineering development can be viewed as a set of unsolved problems. Over time, the number of problems remaining to be solved, P(0), decreased exponentially to zero (as shown in Figure 5-1).

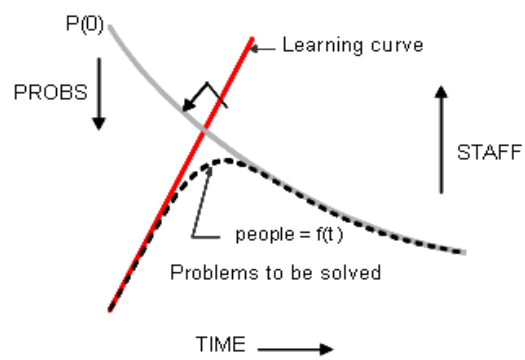


Figure 5-1: Rayleigh-Norden project staffing profile

Norden also noted that the rate at which the project could effectively absorb staff was proportional to the complexity of the problem to be solved. This rate is noted in the figure as the "learning curve." History validates the heuristic that the more complex the problem, the slower the rate that staff can be added to (or absorbed by) the project. Combining the two curves produced the curve shown in the figure as a dashed line (often referred to as the Rayleigh-Norden Curve). The dashed line represents the optimum staffing profile in a software development project.

In probability theory and statistics, the **Rayleigh distribution** is a continuous probability distribution. It can arise when a two-dimensional vector (e.g. wind velocity) has elements that are normally distributed, are uncorrelated, and have equal variance. The vector's magnitude (e.g. wind speed) will then have a Rayleigh distribution.

[23] Norden, P. "Useful Tools for Project Management." Management of Production. Penguin Books. New York, NY: 1970.

In an ideal software development, the peak of the staffing curve occurs at the time of the Formal Qualification Test (FQT), or the end of full-scale software development. Optimum staffing increases during the entire development process to minimize the development schedule. The decaying portion of the staffing curve occurs during system integration and operations.

The Rayleigh-Norden staffing profile is described by the relationship

$$f(t) = (K / t_d^2) t \exp(t^2 / 2t_d^2) \qquad (5\text{-}3)$$

where $K = (E_d / 0.3945)$,

E_d = full scale development effort (person-years),

t = elapsed development time (years), and

t_d = full scale development time (years).

The value of K / t_d^2 defines the maximum staffing rate in persons per year. Exceeding the maximum staffing rate correlates strongly with project failure.

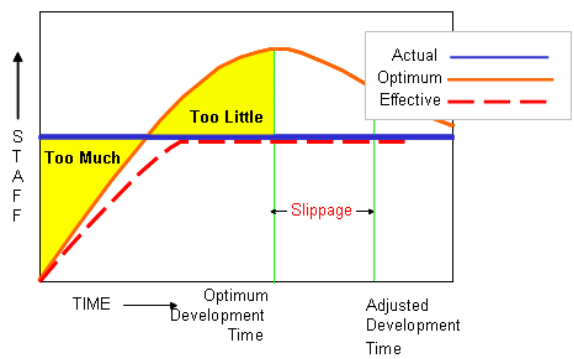


Figure 5-2: Effects of improper staffing

The effects of over- and under-staffing are illustrated in Figure 5-2. Front-loading the staffing profile – in an attempt to decrease development schedule – adds people to the project at a rate that exceeds the rate the project can absorb them. The result is wasted effort that is not only unavailable to the project, but also takes people from the project that could be performing useful work. If the project is planning the development effort to be E_d person months, funding will not be available for the staff needed to maintain the Rayleigh-Norden profile, so the ultimate effect will be potentially higher costs and a delivery delay or slipped schedule. The Rayleigh-Norden staffing profile results in the optimum development schedule and is the most efficient project staffing method.

Brooks's law is a principle in software development which says that adding manpower to a software project that is behind schedule will delay it further. It was coined by Fred Brooks in his 1975 book, *The Mythical Man-Month*.

5.2 Product complexity

An important parameter in both system-level and component-level estimates is the product complexity. Complexity is determined for both estimate types in the same manner as described in Section 4, so the methodology is not repeated here.

5.3 Size estimating process

The size values used in software cost and schedule estimates are derived from a three-step process:

1. Determine the baseline effective size value and the baseline size standard deviation for each CSCI. If the system contains function points as part of the size specification, the function points (including minimum, most likely, and maximum counts) must be converted into equivalent (effective) source lines of code as part of this step. Some estimating tools allow the input of function points as a size measure. The function point count is converted internally to equivalent source lines of code using a process known as "backfiring."

2. Determine the maximum size growth for each CSCI. The maximum growth is determined from the developer capability, the maturity of the software product, the product complexity, and the reuse level of pre-existing source code.

Note: equivalent source lines related to total code size, not effective code size.

3. Determine the mean growth estimate for each CSCI. This value is normally used as the most likely size in the majority of cost and schedule estimates.

This process is identical to that described for system-level estimates in Section 4.5, so it is not repeated here.

5.4 Development environment

The first step in evaluating the development environment for the estimate is to determine the developer's raw capability; that is, the capability without the degradation or loading caused by the environment and constraints imposed by the project requirements. This measure normally changes very little over time unless there is a major change in the corporate approach to software development. This rating is described in more detail in Section 8 and Appendix I. The basic technology constant equation is contained in Section 5.4.4.

The basic capability ratings are consistent across the majority of software development organizations and are close to the mean basic technology constant rating for the industry.

5.4.1 Personnel evaluation

The first environment set is the experience level of the personnel within that environment. There are four experience categories of interest: development system, development practices, target system, and the required programming language.

A segment of the environment set is the management factors. They have an impact on productivity. These factors reflect the use of personnel from other organizations and the dispersal of development personnel across multiple sites by the development organization. The combined impact (product) of these parameters can produce a substantial productivity penalty. These ratings are described in more detail in Section 8.

5.4.2 Development environment evaluation

The second environment segment evaluation determines the impact of the stability and availability of the environment; that is the volatility of the development system and associated practices, the proximity of the resource support personnel, and access to the development system.

For example, a development environment might be described as follows: the development practices are expected to experience minor changes on a monthly average during development due to process improvement. Process improvement should ultimately improve productivity, but will decrease productivity in the short-term during the period of change. The development system will also show productivity losses during system updates that occur on a monthly average. This is a normal condition. The ideal approach is to freeze the system hardware and software during the duration of the project.

Detailed descriptions of the development environment ratings are provided in Section 8.

5.4.3 Product impact evaluation

The third environment segment is used to evaluate the impact of the product characteristics and constraints on the project productivity. The information in this segment is reasonably constant for a given product type, and can be

combined into a template that may experience few changes within the product type due to specific product implementations.

Requirements volatility is a challenging parameter to discuss and evaluate. The volatility definition we use in the guidebook is: the software is a "known product with occasional moderate redirections," since the product will have some functionality improvements beyond the existing products.

The individual product characteristic parameter ratings are described in more detail in Section 9.

5.4.4 Basic technology constant

The basic technology constant[24] C_{tb} describes the *developer's raw capability* unaffected by the project environment. The basic technology constant is calculated using the algorithm in Equation (5-4):

> *People under time pressure don't work better; they just work faster.*
>
> DeMarco and Lister

$$
\begin{aligned}
T &= ACAP * AEXP * MODP * PCAP * TOOL * RESP \\
v &= -3.70945 * \ln\left(\frac{T}{4.11}\right) \\
a &= \frac{v}{5 * TURN} \\
C_{tb} &= 2000 * e^{a}
\end{aligned}
\tag{5-4}
$$

where *ACAP* is the measure for analyst capability,

AEXP is a measure of developer application experience,

MODP is the use of modern development practices,

PCAP is the programmer capability,

TOOL is use of modern development tools,

RESP is the development system terminal response time, and

TURN is the development system hardcopy turnaround time.

The parameters *ACAP*, *AEXP*, *MODP*, *PCAP*, *RESP* and *TURN* are a subset of the environment factors described in detail in Section 8.

5.4.5 Effective technology constant

The productivity factor for Jensen-based models (Seer, SEER-SEM™ and Sage) is the effective technology constant C_{te} defined in Equation (5-5). This equation combines the basic technology constant with the product of the development environment and product characteristic factors f_i :

[24] Jensen, Dr. Randall W. An Improved Macrolevel Software Development Resource Estimation Model. Proc. of the Fifth Annual International Society of Parametric Analysts Conference. St. Louis, MO, Apr. 26-28, 1983.

$$C_{te} = \frac{C_{tb}}{\prod_i f_i} \qquad (5\text{-}5)$$

where C_{te} is the effective technology constant,

C_{tb} is the basic technology constant defined in Sections 5.4.4 and 8, and

f_i are the remaining environment factors not included in the basic technology constant calculation. These are covered in Sections 8-10 and Appendix I.

The effective technology constant is equivalent to the productivity factor introduced in Sections 4 and 7. The constant combines the impact of the development environment, which is not captured in the productivity factor, and the product constraints that are captured in the 13 application types. A 2:1 ratio of the basic to the effective technology constant is common and a ratio of 4:1 is not uncommon for highly constrained or secure software systems.

The productivity index (PI) is another widely used form of an effective technology constant. The PI used by the Software Life Cycle Model (SLIM®) estimating approach from Quantitative Software Management (QSM) is described in detail in Section 7.2.2.

5.5 Development cost and schedule calculations

Seer is an implementation of a component-level model that will be used to demonstrate cost and schedule estimates. This model was created in 1979 and is the basis of the Sage and SEER-SEM™ (now SEER for Software™) software estimating tools described in Appendix E.

Equation (5-6) is the Seer representation of the general component-level effort equation shown in Equation (5-1). The product of the environment factors is calculated in the denominator of the bracketed portion of Equation (5-6). Note that *D* (complexity) is factored out of the product to illustrate that development cost is a function of complexity in this model. The equation shows that the simpler the project (higher value of *D*), the higher the effort, when given the size and the environment. Conversely, the higher the complexity, the lower the effort; however, the schedule is increased, as shown in Equation (5-7). This relationship pair is referred to as the cost-schedule tradeoff. A longer schedule means a smaller team which, in turn, equates to higher productivity:

$$E_d = 4.72\left[\frac{D^{0.4}}{C_{te}^{1.2}}\right]S_e^{1.2} \text{ person months} \qquad (5\text{-}6)$$

where E_d is the full-scale development effort (PM),
D is the product complexity,
C_{te} is the effective technology constant, and
S_e is the effective component (CSCI) size (ESLOC).

Equation (5-7) specifies the development schedule in months.

$$T_d = \left[\frac{12}{C_{te}^{0.4} D^{0.2}}\right] S_e^{0.4} \text{ months} \qquad (5\text{-}7)$$

where T_d is the full-scale development schedule (months),
C_{te} is the effective technology constant,
D is the product complexity, and
S_e is the effective component (CSCI) size (ESLOC).

Equation (5-8) is an alternate form of the schedule equation that is consistent with the general schedule form shown in Equation (5-2). Note again that by factoring D out of the proportionality constant, the effect of D on the schedule can be seen explicitly:

$$T_d = \left[\frac{7.15}{\sqrt[3]{D}}\right] \sqrt[3]{E_d} \qquad (5\text{-}8)$$

where T_d is the full-scale development schedule (months),
E_d is the full-scale development effort (PM), and
D is the product complexity.

5.6 Verify estimate realism

Once an estimate is complete, it is important to step back and objectively validate the results. The most important estimate reality checks are:

1. Is the productivity value for each component reasonably close to the productivity value for that component type (see Table 7-1)?
2. Is the productivity estimate comparable to historical developer productivity for similar projects?
3. Is the size, including growth, for each component within the range of 5,000 to 200,000 source lines?
4. Is the size estimate, with growth, consistent with the effective size for completed historical projects?
5. Is the schedule for each component greater than $2.8 \times \sqrt[3]{E_d}$?

All estimates should be validated. Most of the historic data we look at is based on the effort (cost) and schedule for the development of CSCIs. An alternate approach that may be used, when historic data is not available, is comparing your estimate's productivity to the factors in Table 7-1. The CSCI of interest is compared to the 13 system types. The software types in your estimate may not map directly to the task names of those listed in Table 7-1; however, the productivity values are typical of the product type. A comparison to the general category is usually sufficient to validate your productivity estimate.

5.7 Allocate development effort and schedule

5.7.1 Effort allocation

The next step in a component estimate is to assign the effort to the development activities. The distribution of effort varies with the effective size of the system. The larger the system, the more effort will be spent in system test and integration. The relative effort spent in creating the architecture increases with size as well. The cost estimation tools (each using their own allocation percentages and equations) normally perform the total project effort allocation for the system.

If you want to calculate the total project effort yourself, use the rough effort distribution as a function of source size presented in Table 5-2.

Table 5-2: Total project effort distribution as a function of product size

Size (KESLOC)	Activity				
	Rqmts. (%)	High-Level Design (%)	Develop. (%)	Sys. Test (%)	Mgt. (%)
1	4	12	77	17	11
25	4	20	67	24	13
125	7	22	63	25	15
500	8	25	56	32	19

Adapted from McConnell, 2006; Putnam and Myers, 1992; Jones, 2000; Boehm et al, 2000; Putnam and Myers, 2003; Boehm and Turner, 2004; Stutzke, 2005.

Note that the design, develop, and management columns add up to 100 percent. However, each row of the table adds up to a percentage greater than 100 percent since the effort allocations are derived from the full scale development effort. Full-scale development includes high-level design (architecture), development (detail design, code and unit test), and internal component integration and management. Hence, the total effort includes full-scale development, requirements analysis, and system test.

Larry Putnam is one of the pioneers in software cost and schedule estimating. He created the SLIM® estimating model in 1976 based on project data obtained from the U.S. Army Computer Systems Command.

The total effort is computed from the relationship:

$$E_{Total} = k \times E_{Develop} \qquad (5\text{-}9)$$

where E_{Total} is the total project effort through system integration (PM)

k = 1.21 to 1.40 (an additional 21 to 40 percent effort) depending on the difficulty of integrating the software component and the system, and

$E_{Develop}$ is the project development effort from SRR through FQT.

A commercial standalone software product development may not require any system integration. Even so, an integration factor value of 1.28 is a reasonable default for CSCI-system integration.

Most component-level software estimating tools perform the effort distribution automatically following an allocation table which may differ from the one presented in Table 5-2. They generally follow the optimum allocation algorithm based on the Rayleigh-Norden staffing profile. Many software estimating tools allow the user to specify the effort allocation for requirements analysis and system test. Table 5-2 is a good guideline for selecting the appropriate allocations for these activities.

5.7.2 Schedule allocation

Schedule allocation depends on the effective size of the software system *and* the software development approach. A classic waterfall development approach has schedule milestones at different times than an incremental or agile development. There are some guidelines to help approximate milestones in ballpark estimates. The cost estimation tools (each using their own allocation percentages and equations) normally perform the schedule allocation for the system.

If you want to calculate the schedule yourself, use the effort schedule breakdown shown in Table 5-3.

Table 5-3: Approximate total schedule breakdown as a function of product size

Size (KESLOC)	Activity			
	Requirements (%)	High-Level Design (%)	Development (%)	System Test (%)
1	6-16 (6)	15-25 (20)	50-65 (55)	15-20 (19)
25	7-20 (7)	15-30 (19)	50-60 (52)	20-25 (22)
125	8-22 (8)	15-35 (18)	45-55 (49)	20-30 (25)
500	12-30 (12)	15-40 (15)	40-55 (43)	20-35 (30)

Sources: Adapted from McConnell, 2006; Putnam and Myers, 1992; Boehm et al, 2000; Putnam and Myers, 2003; Stutzke, 2005.

The overall schedule distribution for any project is based on the entire project totaling 100 percent of the schedule. The numbers in parentheses represent typical allocations for each size group. Judgment is necessary when selecting the specific values from the table for the project in question.

5.8 Allocate maintenance effort

The term "life cycle cost" has multiple meanings. A common definition relates to the software development process which includes all of the development phases from requirements analysis through system integration. Another realistic definition includes all of the phases from requirements analysis to disposal at the end of the software's operational life. The second definition includes maintenance of the delivered software product, which can be many times the cost of development[25]. Historical data shows that it is not uncommon for the cumulative cost of maintenance to exceed the cost of full-scale development by the end of the fifth calendar year of operations.

The effort related to the maintenance categories is discussed in Section 4.8. The calculations and procedures are common to both system- and component-level estimates. Note that the component-level estimating model can be used to predict the defect removal effort unless there is adequate information describing the number of defects introduced during development; however, there is usually enough information to estimate the enhancement and operation support costs.

[25] Another widely used, inappropriate definition for maintenance is "when the development budget is depleted." This is not a definition used in software estimating, but included for completeness.

Software is like entropy. It is difficult to grasp, weighs nothing, and obeys the Second Law of Thermodynamics; i.e., it always increases.

Norman Augustine, 1983

Section 6

Estimating Effective Size

Effective size is the most important cost and schedule driver in a software development estimate. Not only is size the most important driver, it is the most difficult to estimate. Size comes in many shapes and forms, but most of them are of little value in realistic effort estimates.

An accurate size estimate could be obtained in the early days of estimating (the 1960s) by counting source lines of code. The big question in those days related to how a source line of code (SLOC) was defined: either by punched card images or executable statements. As software developments increased in size, the meaning of SLOC became complicated because of the use of reused code in large projects. The definition of SLOC was still not resolved until as late as 2005.

The SLOC definition frustration led many researchers to seek alternate methods of measuring the size of a software development. Albrecht and Gaffney[26] of IBM proposed a functional measure of software size that became a second size standard. Function Points (FPs) are a measure of *total* software size, which ignores reuse, but is effective very early in the development process when SLOC is still difficult to predict. FPs spawned new sizing methods including feature points, object points, and use cases, among others.

This guidebook discusses the two most widely used software-sizing approaches: SLOC and FPs. The approaches are radically different from each other. The discussion of the two methods can be extended to support others that continue to evolve from the SLOC and FP methods. SLOC are the focus of the sections starting with Section 6.1. FPs are the topic of discussion beginning in Section 6.7.

Neither of these approaches solves the problem of estimating the effective size necessary to project the cost and schedule of a software development.

The early 2000s introduced the use of code counters as a means of obtaining size information from completed software components. This allowed the gathering of computer software configuration item (CSCI) or functional sizing data. Depending on the configuration of the counting software, the measured size could represent executable SLOC, total SLOC, or something in-between. This data can help reduce the difficulty in estimating cost and schedule.

Size information includes three physical components:

1. Amount of new source code to be *added* (new) to satisfy current product requirements.
2. Amount of the reused source code to be *modified (changed or deleted at the module level)* to satisfy current product requirements.
3. Amount of source code being *reused* without change from previous software products.

---

[26] Albrecht, A. J., and Gaffney, J. E. "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation." IEEE Transactions on Software Engineering. Nov. 1983.

Effort is not simply related to the number of SLOC, but is related to the work required to use or produce those lines. Reused SLOC is not a zero-effort component because work is required to incorporate and test that component.

Assuming a constant effort for size is equivalent to assuming the development includes only new code. That is not a good assumption; the effort needed to modify and reuse code is different than creating all new code.

A definition of source code requires some base architecture definitions. The CSCI is the fundamental element for which effort, schedule, and productivity data is collected in universally applicable numbers. Lower-level data is very dependent upon the individual performance and, therefore, not generally applicable to estimates. System-level data also is difficult to use because of the combination of different complexity types of the CSCIs.

CSCIs contain elements that can be major subcomponents of the CSCI or as small as modules (procedures, subroutines, etc.). There can be several component layers in the CSCI, of which the smallest element is the module (or unit) (see Figure 6-1). Modules contain SLOC including executable source lines of code, declarations, directives, format statements, and comments.

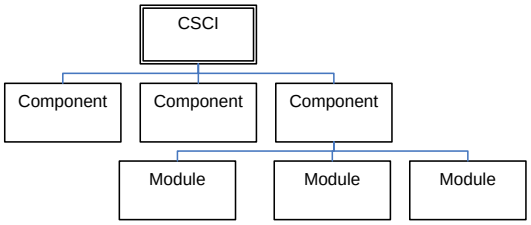


Figure 6-1: Source code taxonomy

The purpose of Section 6 is to demonstrate the philosophy and process of developing an effect size value that can be used in a realistic software cost and schedule estimate.

6.1 Source code elements

A line of source code is defined as a program statement that consists of new, modified, deleted, or reused code or commercial off-the-shelf (COTS) software. Total size includes all four statement types. The elements are defined in subsequent subsections. First, we need to define some important concepts: the black box and the white box.

6.1.1 Black box vs. white box elements

A black box is a system (or component, object, etc.) with known inputs, known outputs, a known input-output relationship, and *unknown* or irrelevant contents. The box is black; that is, the contents are not visible. A true black box can be fully utilized without knowledge of the box content.

A white box is a system that requires user knowledge of the box's contents in order to be used. Component changes may be required to incorporate a white-box component into a system. The box is white or transparent, and the contents are visible and accessible.

A software component becomes a white box when any of the following conditions exist:

1. A component modification is required to meet software requirements.
2. Documentation more extensive than an interface or functional description is required before the component can be incorporated into a software system.

The black box concept, applied to software systems, becomes particularly important where black boxes (or objects, reusable components, etc.) are used as components without the engineering, implementation, integration, and test costs associated with their development. COTS software is an example of an applied black box. A white box has visible source code; that is, the total

accessible source code size in the box is greater than zero. Black box code is unavailable and unknown; that is, the total source code size within the black box is conceptually equal to zero (as shown in Figure 6-2). No knowledge of the black box code is necessary to utilize the black box component.

COTS software elements satisfy the black box definition. There may be millions of executable source code lines within the black box, but knowledge of this internal code is not necessary to utilize the code in a new system. Only terminal characteristics are required for use.

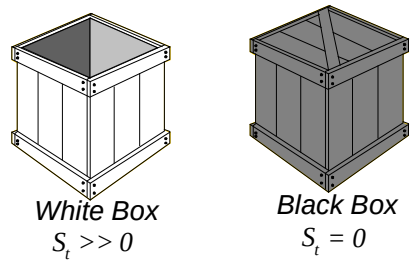


Figure 6-2: Black box description depicting relationship between effective size (S_t) and box type.

6.1.2 NEW source code

Source code that adds new functionality to a CSCI is referred to as "new" code. New code, by definition, is contained in new modules. Source code modifications packaged in reused (pre-existing) modules are referred to as "modified" source code.

6.1.3 MODIFIED source code

Source code that changes the behavior or performance of a CSCI is referred to as "modified" code. Modified code, by definition, is contained in pre-existing white-box modules. If one countable SLOC within a module is added, deleted, or modified; the entire countable SLOC in the module is counted as modified SLOC. For example, if a module contains 60 SLOC of which we delete 10 SLOC and change 10 SLOC, the modified SLOC count is 50 SLOC (60 – 10).

6.1.4 DELETED source code

Source code that deletes functionality from a CSCI is referred to as "deleted" code. Deleted code, by definition, represents the total source code contained in a deleted module. Source code deletions within a modified module are *not* counted as deleted source code.

6.1.5 REUSED source code

Modules that contain executable, declaration, directive, and/or form statements (satisfies the design, document, and test criteria), and are used without modification are referred to as "reused" code. Like modified code, reused code is also contained in pre-existing white-box modules. Reused code is not modified in any way, but some analysis and/or testing may be necessary in order to assess its suitability for reuse. The countable source code in the reused module is known; that is, the total size is greater than zero. It is possible to reverse-engineer and perform regression tests on reusable elements.

COTS products have undefined size, only functional purposes; therefore, you cannot include COTS software as part of the reuse size. Effort must be added directly in the effort computations.

6.1.6 COTS software

It is important to formally define COTS software because of its impact on the physical and effective size of the software CSCI. This definition is clearly described using the definitions of black and white boxes.

Software black box component (COTS) behavior can be characterized in terms of an input set, an output set, and a relationship (hopefully simple) between the two sets. When behavior requires references to internal code, side effects, and/or exceptions, the black box becomes a white box.

COTS software is black-box software for which the countable source code is not known, only the functional purpose. Since the contents of the COTS element are unknown (total SLOC is essentially zero, $S_t = 0$), the COTS element can only be tested from the element interface as part of the system. Reverse-engineering and regression testing cannot be performed on a COTS element.

6.1.7 Total SLOC

Total SLOC is the sum of the modified, the reused, and the new SLOC. As indicated above, the contribution of COTS to the total SLOC is zero.

6.2 Size uncertainty

Size information early in a project is seldom, if ever, accurate. Information presented in a Cost Analysis Requirements Description (CARD) is generally predicted during a concept development phase prior to the start of full-scale development. CARD information contains considerable uncertainty. Size information is often presented as a single value which leads to a point estimate of cost and schedule; however, the information is more accurately expressed as three values representing a minimum (most optimistic), a most likely, and a maximum (most pessimistic) value to give a more realistic picture of the information.

The size values used in a software cost and schedule estimate are typically derived from a three-step process. The first step determines the nominal effective, or most likely, size value and the nominal size standard deviation. The "low" size value (10 percent probability) is equal to the mean value minus the nominal size standard deviation in the normal distribution shown in Figure 6-3. In practice, the size distribution is skewed (non-normal).

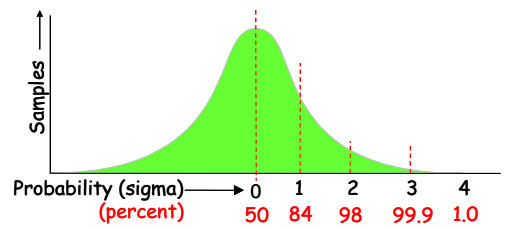


Figure 6-3: Normal distribution

The second step produces an estimate for the maximum size (90 percent probability).

The third step determines the mean growth estimate used as the most likely size value in the cost and schedule estimate. The mean estimate, assuming a normal distribution, is the center value of the distribution (as shown in Figure 6-3). A non-normal mean value will be above the median value for a distribution, skewed toward the low side. The low, mean, and high size values are also used in the software risk analysis.

As an example, the CSCI size could be at least 5,000 source lines, most likely 7,500 lines, or as large as 15,000 lines. The mean value is not necessarily in the center of the range; that is, the peak of a normal distribution. The available commercial estimating tools internally adjust the mean value of the size distribution to a pseudo-normal form with an associated standard deviation. The mean value formula is:

$$S_{mean} = (S_{min} + 4S_{ml} + S_{max})/6 \qquad (6\text{-}1)$$

where S_{mean} = mean size value,
S_{min} = minimum size value,
S_{ml} = most likely size value, and
S_{max} = maximum size value.

and the standard deviation is

$$S_{std} = (S_{max} - S_{min})/6 \qquad (6\text{-}2)$$

6.3 Source line of code (SLOC)

During the summer of 2006, a group of senior software estimators from major software developers in the aerospace community formulated a SLOC definition that was compatible with the commercial software estimating tools and the major estimating methodologies. The definition is also compatible with current code counting methods. This section contains the result of the 2006 collaboration.

SLOC is a measure of software size. A simple, concise definition of SLOC is any software statement that must be ***designed***, ***documented***, and ***tested***. The three criteria (designed, documented, and tested) must be satisfied in each counted SLOC. A statement may be long enough to require several lines to complete, but still is counted as one SLOC. Multiple statements may be entered on a single line, with each statement being counted as a single SLOC. All source statements can be categorized into one of five categories:

1. Executable
2. Data declarations
3. Compiler directives
4. Format statements
5. Comments

Comment statements do not satisfy the SLOC criteria. Comments are not designed, documented, or tested, but are part of the software documentation. Applying the SLOC criteria to the remaining four statement categories, the following statements are referred to as ***countable*** SLOC:

- Executable
- Data declarations
- Compiler directives
- Format statements

Each of the countable SLOC categories is defined in the following subsections.

6.3.1 Executable

Executable statements define the actions to be taken by the software. These statements do not include logical *end* statements (fails SLOC criteria) or debug statements inserted by the programmer for development test unless the debug code is required in the software requirements. In other words, non-deliverable debug code is not formally designed or tested, and likely not documented either—and therefore ignored in the executable SLOC count.

6.3.2 Data declaration

Data declarations are formally designed, documented, and tested when they are written (or defined). Declarations are commonly copied into software elements once they have been defined.

6.3.3 Compiler directives

Compiler directives are formally designed, documented, and tested when they are written. *Include* statements for the data definitions are each counted as one SLOC where entered.

6.3.4 Format statements

Format statements are remnants of the Fortran and COBOL programming languages, but do satisfy the criteria for definition as a SLOC. Format statements often require multiple lines for definition.

6.4 Effective source lines of code (ESLOC)

A realistic size estimate is necessary to reasonably determine the associated software development cost and schedule. Physical source instructions *include* all instructions that require design, documentation, and test. Debug statements emphasize the importance of this source instruction definition; for example, debug statements are formally designed, documented, and tested only if required by the development contract. Non-required debug statements are a normal development product needed only to produce other source instructions. Non-required debug statements are not designed, documented, or thoroughly tested; thus, they are *excluded* from the source line count.

Resource estimates based on physical source lines for modified software, and systems containing reusable components, cannot account for the additional resource demands attributable to reverse-engineering, test and software integration. The common method used to account for the added resource demands is to use the *effective* software size.

6.4.1 Effective size as work

Work is a useful and necessary concept when applied to defining effective size for cost calculation. Development SLOC are numerically identical to physical source lines when the total software product is being developed from scratch or as new source lines. Most software developments, however, incorporate a combination of new source code, existing code modified to meet new requirements, unmodified existing code, and off-the-shelf components. Physical SLOC, as defined in this guidebook, do not relate directly to the development or effective size. Effective size is larger than the actual change size; that is, development size is equal to, or greater than, the sum of new and modified source lines. Effective size is usually smaller than the total physical source lines in the software product or there would be no incentive to utilize reusable elements in software development.

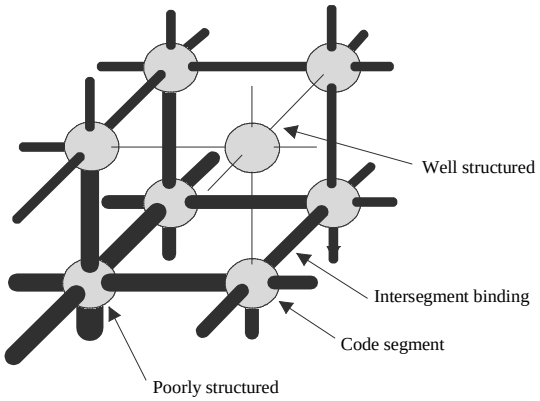
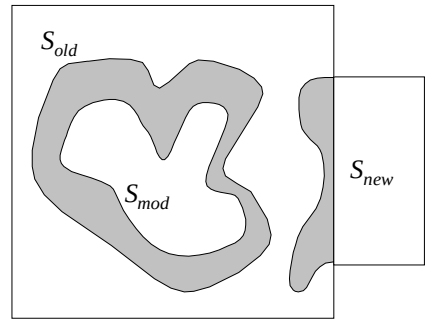


Figure 6-4: Molecular view of software demonstrates the impact of structure on effective size

The molecular software view in Figure 6-4 illustrates the need for a mechanism to determine the effective software size (S_e). The *ideal*

inter-molecular coupling between software elements, be they elements, segments, blocks, objects, or any other lumping, is loose. Loose coupling—the goal of structured design, object-oriented design, and most other design approaches—makes it possible to modify or insert elements without reverse-engineering or testing portions of the software system that are not to be modified. The other coupling extreme (tight coupling) occurs in what we refer to as "spaghetti" code. Tight coupling requires complete reverse-engineering and integration testing to incorporate even modest software changes.

The relationship between size elements and the size adjustment factors is illustrated in Figure 6-5. The shaded portion around the additions and modifications in the figure represent the software system areas that must be reverse-engineered, and tested, but are not modified.



$$S_{tot} = S_{old} + S_{new}$$
$$S_{chg} = S_{new} + S_{mod}$$

Figure 6-5: Development effort required to incorporate changes to an existing software system includes efforts to produce S_{new}, S_{mod} and integration effort in peripheral shaded areas

Effective task size must be greater than or equal to the number of source lines to be created or changed. Modified software systems require accounting for the following cost impacts not present in new systems:

1. Modifications cannot be made to a system until the engineers and programmers understand the software to be modified. If the original developer makes the changes, modifications are much simpler than if the developer is unfamiliar with the architecture and source code. The software architecture determines the reverse-engineering required to implement changes. Clear, well-organized documentation also reduces the system learning curve and the required reverse-engineering. *The rationale for counting the total executable lines in a modified module (50-200 SLOC), instead of the few physical lines changed, is the effort impact of the change is much greater than the physical change unless the module is well structured and documented and the module is being modified by the originator*.

2. The effort required to implement (code) software modifications must include effort to correctly implement interfaces between existing software and the modifications. This requires a level of knowledge in addition to that discussed in the first point.

3. All software modifications must be thoroughly tested. The test effort in utopian systems consists of testing no more than the software changes. In reality, total system regression tests are required prior to delivery to ensure correct software product performance. It is virtually impossible to reduce integration and test requirements to include no more than software modifications.

4. Although it is not obvious from the discussion to this point, the effective size must also be adjusted to account for the development environment; that is, the quality of the developer, experience with the software, and the experience with the development environment.

Resource estimates based on physical source lines for modified software and systems containing reusable components cannot account for the additional resource demands attributable to reverse engineering, test, and software integration. Effective size is the common method used to account for the added resource demands.

Each of these four activities increases effective task size and is manifested as increased cost and schedule. Major estimating systems increase development size to adjust for the added effort requirements. COCOMO[27] defines this adjustment term, the Adaptation Adjustment Factor (AAF), as:

[27] Boehm, B.W. Software Engineering Economics Englewood Cliffs, NJ.: Prentice-Hall, Inc., 1981.

$$AAF = 0.4F_{des} + 0.3F_{imp} + 0.3F_{test} \qquad (6\text{-}3)$$

where

F_{des} = the fraction (percent redesign [%RD], see Section 6.4.2.1) of the reused software requiring redesign and/or reverse-engineering,

F_{imp} = the fraction (percent reimplementation [%RI], see Section 6.4.2.2) of the reused software to be modified, and

F_{test} = the fraction (percent retest [%RT], see Section 6.4.2.3) of the reused software requiring development and/or regression testing.

The simplified Jensen effective size[28] uses a similar adjustment equation, referred to as the Size Adjustment Factor (SAF):

$$SAF = 0.4F_{des} + 0.25F_{imp} + 0.35F_{test}\,. \qquad (6\text{-}4)$$

Equation (6-4) places greater emphasis on the development cost and schedule impact of the test activity than the relative impact captured in Equation (6-3). The 40-25-35 division of effort is consistent with historic data and closer to the traditional 40-20-40 division of the development effort heuristic.

A simple example of the AAF factor in practice is as follows:

> Assume a software module containing 50 executable SLOC. The original module was implemented by an outside organization. The module's internal algorithm is poorly structured and documented (normal), and lacking test data and procedures. The module has been modified by a third organization. The planned use of this module requires only small code changes to satisfy current system requirements. Your use of this module will require reverse-engineering (redesign), the modification is assumed to involve only 10 percent of the module (reimplementation), and testing (modifications and regression). The resulting values for the design, implementation, and test factors are essentially 1.0. Thus, the AAF value is also 1.0.

6.4.2 Effective size equation

Effective size is generally defined by one of the two relationships used for computing the effective size or effective source lines of code (ESLOC). The terms in the two relationships must satisfy the source code criteria.

$$S_{effective} = S_{new} + S_{\text{mod}} + S_{reused}\,SAF \qquad (6\text{-}5)$$

where S_{new} = new software SLOC,

S_{mod} = modified software SLOC,

S_{reused} = reused software SLOC, and

SAF = Size Adjustment Factor (Equation 6-4).

---

[28] Jensen, R.W. A Macrolevel Software Development Cost Estimation Methodology. Proc. of the Fourteenth Asilomar Conference on Circuits, Systems, and Computers. Pacific Grove, CA. Nov. 17-19, 1980.

$$S_{effective} = S_{new} + S_{\text{mod}} + S_{reused} AAF \qquad (6\text{-}6)$$

The first equation (Equation 6-5) is used by all Jensen-based (Seer, SEER-SEM™ and Sage) estimating tools.

The second equation (Equation 6-6) is used by all COCOMO-based (COCOMO I, REVIC, COCOMO II) estimating tools.

The effective size values computed by both methods are approximately equal. The first equation is used for consistency in the software data collection form described in Appendix I.

The following subsections explain the elements of the AAF and SAF equations in greater detail.

6.4.2.1 Design factor

The design factor F_{des} is the ratio of the reverse-engineering effort of S_{reused} required for the software development to the amount of work that would be required to develop S_{reused} from scratch. A reasonable F_{des} approximation can be obtained from the ratio of the number of routines (procedures, subroutines, etc.) to be reverse-engineered to the total number of routines in S_{reused}.

6.4.2.2 Implementation factor

Implementing the software system changes in Figure 6-5 requires S_{new} + S_{mod} be produced. Additional effort must be added to understand the interfaces between the changes (enhancements and modifications) and S_{reused}. The extra effort includes establishing calling sequences, the number of parameters, the units assumed by each parameter, etc. Thus, a reasonable value for F_{imp} is the ratio between the interface effort and the effort required to produce the interface definitions from scratch. The interface effort is generally a small number (less than 10 percent).

6.4.2.3 Test factor

The F_{test} factor is the ratio of the S_{reused} test effort anticipated through acceptance of the upgraded software system to the effort required to test the reused code from scratch. Regression tests (acceptance test, etc.) are necessary to ascertain correct system performance with the incorporated upgrade. This reusable effort includes test and integration plans, procedures, data, and documentation.

Example – Effective computation with reusable components

Assume a reasonably well-structured software product is being upgraded by the original developer. The product has the following size attributes and an average module size of 2,000 SLOC:

S_{new} = 20,000 SLOC (10 procedures)
S_{mod} = 40,000 SLOC (20 procedures)
S_{reused} = 100,000 SLOC (50 procedures)

Software reverse-engineered

without modification (10 procedures)

Test effort (including test plan and procedure development), salvageable from previous product versions, is 25 percent of total test effort of S_{reused}. The additional effort required to properly interface the software enhancements and modifications is 1 percent of S_{reused}.

$$F_{des} = \frac{10}{50} = 0.2$$

$$F_{imp} = 0.01$$

$$F_{test} = 1.0 - 0.25 = 0.75$$

$$S_e = 60{,}000 + 100{,}000\ [0.4(0.2) + 0.25\ (0.01) + 0.35\ (0.85)]$$

$$= 94{,}500 \text{ ESLOC}$$

To demonstrate the effective size impact of developer software product experience, let us assume the original software product was poorly documented and built by a competing contractor. To adjust for the increased reverse product engineering, we change the F_{des} factor to 1.0 (reverse-engineer the entire product), and change F_{test} to 1.0 (no access to the original test material). The resulting effective size is:

$$S_e = 60{,}000 + 100{,}000\ [0.4(1) + 0.25\ (0.01) + 0.35\ (1)]$$

$$= 135{,}250 \text{ ESLOC}$$

The effective source increase due to lack of developer product familiarity is 40 percent over the original 94,500 ESLOC estimate due to environment changes.

6.5 Size growth

Some experts consider software code growth to be the single most important factor in development cost and schedule overruns. Code growth is caused by a number of factors: requirements volatility, size projection errors, product functionality changes, and human errors.

Size projection errors come from many sources. Errors may stem from a lack of historical size data. Errors can also arise from a need to keep the development cost low in order to obtain project funding or to submit a winning proposal by a contractor. Size is the primary cost driver in the development estimate. Figure 6-6 shows samples of size growth from several major software developments (represented by A – K on the x-axis). The reasons for growth in these projects are not available, which complicates the development of any growth model.

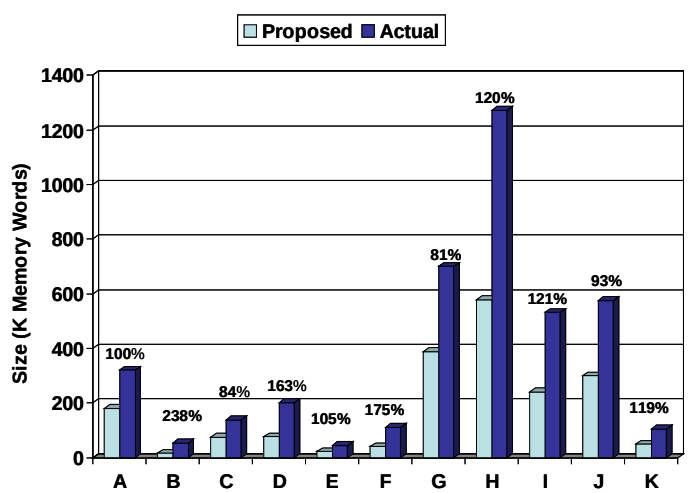


Figure 6-6: Historic project data basis for growth algorithm

Size errors can also arise from lack of experience and the simple human limitations of the estimator. One reason dates back to Greek mythology and the legend of

Pandora. According to the myth, Pandora opened a jar (pithos) in modern accounts referred to as "Pandora's box", releasing all the evils of mankind— greed, vanity, slander, envy, lust— leaving only one evil inside once she had closed it again. When Pandora opened the jar the second time, she unleashed the worst of all afflictions on the human race -- Hope. Because of hope we don't learn from our mistakes and, consequently, are forever cursed with unbounded optimism.

Another reason is the human mind cannot wrap itself around a large number of independent objects. Various sources place the limits anywhere from 7 +/- 2 to as few as 3 +/- 2 objects. Thus, details get buried or hidden in our projection of the software required to perform a complex task. Software size is almost always underestimated, and the magnitude of this underestimate is usually not trivial.

There have been several studies on software code growth published during the past 15 years. "Several" does not necessarily equate to enough. The data from the most recent of these studies[29] is typical for embedded software system development. The growth range for these projects is between 81 percent and 238 percent.

Watts Humphrey[30], one of the driving forces behind the Capability Maturity Model (CMM), states that while there is no good data to support these figures, there are some practical estimating rules of thumb, which are shown in Table 6-1. These rules are consistent with project data, and are reasonably good guidelines.

Table 6-1: Code growth by project phase

Completed Project Phase	Code Growth Range (Percent)
Requirements	100-200
High-Level Design	75-150
Detailed Design	50-100
Implementation	25-50
Function Test	10-25
System Test	0-10

6.5.1 Maximum size growth

Barry Holchin[31], a well-known cost researcher and estimator, proposed a code growth model that is dependent on product complexity (D), project maturity, and the distribution of new and reused source code. The Holchin model provides a mechanism to predict the physical (or total) growth during software development.

The region between the maximum and minimum complexity lines (D_{max} and D_{min}) represents the potential growth region, as shown in Figure 6-7. The values of D_{max} and D_{min} are 15 and 8, respectively. The values are derived from the ratio of the development cost to the development time:

$$D = cE_d / T_d^3 \qquad (6\text{-}7)$$

where D = complexity,
c = scaling constant,
E_d = development effort (person-years), and
T_d = development time (years).

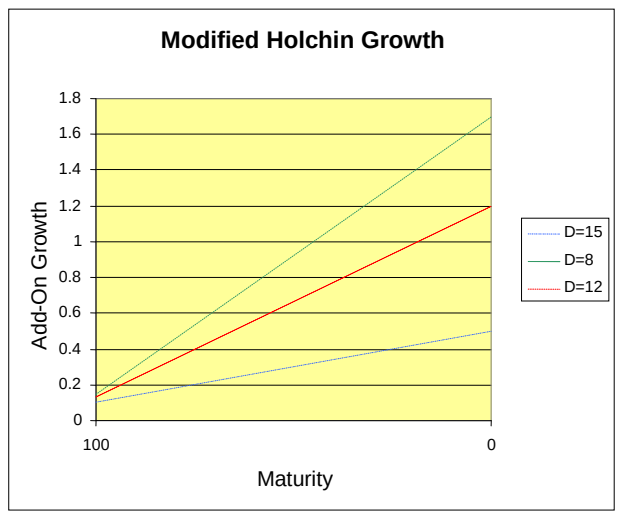


Figure 6-7: Modified Holchin growth algorithm
(Both axes represent percentages)
Top line (D=8), Middle (D=12), Bottom (D=15)

A low complexity (high D value) project allows higher staff levels to be applied to the development over a shorter period. A high-complexity project necessitates a smaller development team over a longer

[29] Nicol, Mike. Revitalizing the Software Acquisition Process. Acquisition of Software Intensive Systems Conf. 28 Jan 2003.
[30] Humphrey, Watts. Managing the Software Process. Addison-Wesley. New York, NY: 1989.
[31] Holchin, Barry. Code Growth Study. 4 Mar. 1996.

period. The complexity value 15 in the Sage, System Evaluation and Estimation of Resources Software Estimating Model SEER-SEM™ (now SEER for Software™), and SLIM® estimating tools equates to the complexity of a standalone software application that does not interact with the underlying operating system. For example, the software (*D =15*) can be an accounting system. A high-complexity project (*D = 8*) equates to an operating system development.

The first step in the growth calculation is to establish the range of the growth factor (shown as *D=12* in Figure 6-7) for the development under consideration. The project complexity rating of 12 (a satellite ground station) represents an application with significant interaction with the underlying system. The growth curve (shown as a dashed line) is calculated from the relationships:

$$C_0 = C_{0\,\min} + (C_{0\,\max} - C_{0\,\min})(D-8)/7 \qquad (6\text{-}8)$$

$$C_{100} = C_{100\,\min} + (C_{100\,\max} - C_{100\,\min})(D-8)/7 \qquad (6\text{-}9)$$

where C_0 = 0 end point of a desired growth curve for an assigned D

C_{0max} = upper bound end point for software growth curve assuming D range of 8 to 15

C_{0min} = lower bound end point for software growth curve

C_{100} = 100 end point of a desired curve for and assigned D

C_{100max} = upper bound end point for software growth curve

C_{100min} = lower bound end point for software growth curve

From an estimating point of view, the physical code growth is important, but not as important as the total size growth. The Holchin model has been extended to project the growth in effective size necessary for development effort calculation. The growth space is, graphically, the same as the space shown between *D = 8* and *D = 15* in Figure 6-7.

Step two of the growth calculation accounts for the project maturity level. The maturity level adjusts the software growth to the current status of the software development. The modified Holchin maturity table is shown in Table 6-2. The maturity factor *M* is applied to the effective size growth equation:

$$h = 1.0 + C_0 + (C_{100} - C_0) * (1.0 - M/100) \qquad (6\text{-}10)$$

where *h* represents the total relative code growth and

M = the maturity factor value.

As an example, if we assume the growth projection is being made at the time of the Software Requirements Review (*D=12* @ SRR), the mean relative code growth of complexity is 1.16, or approximately 16 percent. Maximum growth is 1.55. This growth effect can be seen in Tables 6-3 and 6-4.

The maturity factors shown graphically in Figure 6-7 are summarized for normal incremental complexity values for mean growth in Table 6-3 and Table 6-4 for maximum growth.

Table 6-2: Modified Holchin maturity scale

Maturity	M
Concept	0
Source	15
C/A	33
SRR	52
PDR	67
CDR	84
FQT	100

Table 6-3: Mean growth factors (h) for normal complexity values as a function of project maturity

		Complexity								
Maturity	**M**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**Comments**
Concept	0	1.50	1.45	1.40	1.35	1.30	1.25	1.20	1.15	Start requirements
Source	15	1.43	1.39	1.34	1.30	1.26	1.22	1.17	1.13	Source Selection (Proposal complete)
C/A	33	1.35	1.31	1.28	1.24	1.21	1.18	1.14	1.11	Start full scale development
SRR	52	1.26	1.24	1.21	1.19	1.16	1.13	1.11	1.09	Requirements complete
PDR	67	1.19	1.18	1.16	1.14	1.12	1.10	1.09	1.07	Architecture complete
CDR	84	1.12	1.11	1.10	1.09	1.08	1.07	1.06	1.05	Detail design complete
FQT	100	1.04	1.04	1.04	1.04	1.04	1.03	1.03	1.03	Development complete

Table 6-4: Maximum growth factors (h) for normal complexity values as a function of project maturity

		Complexity								
Maturity	**M**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**Comments**
Concept	0	2.70	2.53	2.36	2.19	2.01	1.84	1.67	1.50	Start requirements
Source	15	2.47	2.32	2.17	2.03	1.88	1.73	1.59	1.44	Source Selection (proposal complete)
C/A	33	2.19	2.07	1.95	1.84	1.72	1.60	1.49	1.37	Start full scale development
SRR	52	1.89	1.81	1.72	1.64	1.55	1.46	1.38	1.29	Requirements complete
PDR	67	1.66	1.60	1.54	1.48	1.42	1.35	1.29	1.29	Architecture complete
CDR	84	1.40	1.36	1.33	1.30	1.26	1.23	1.20	1.16	Detail design complete
FQT	100	1.15	1.14	1.14	1.13	1.12	1.11	1.11	1.10	Development complete

The third step in calculating growth projects the total size growth at the end of development from:

$$S_{TG} = hS_0 \qquad (6\text{-}11)$$

where S_{TG} = total size with growth and

S_0 = projected size before the growth projections.

The size growth projection at this point is for "total" source code size. The size growth projection necessary for an estimate is based on the "effective" source code size. In that regard, the growth must include not just physical growth, but the impact of the software modifications, reverse-engineering, regression testing, etc. The effective size is given by:

$$S_{eff} = S_{new} + S_{\text{mod}} + S_{reused}(.4F_{des} + .25F_{imp} + .35F_{test}) \qquad (6\text{-}12)$$

where F_{des} = relative design factor,
F_{imp} = relative implementation factor, and
F_{test} = relative test factor.

The new source code is specified by S_{new}, the modified source code by S_{mod}, and S_{reused} specifies the reused source code value. There are three more areas that will experience growth, from the effective size point of view:

1. Reverse-engineering of the reused code to define modifications required by the new system; that is, $S_{des} = 0.4F_{des}S_{reused}$.

2. Interfaces to the modified code must be defined as a percentage of $S_{imp} = 0.25 F_{imp} S_{reused}$.

3. Regression testing of the reused code to assure proper performance of the upgraded system; that is $S_{test} = 0.35 F_{test} S_{reused}$.

The effective size growth can be approximated by applying the growth factor *h* to each of the three areas. The effective size growth is then given by:

$$S_{TE} = h(S_{new} + S_{\text{mod}} + S_{des} + S_{imp} + S_{test}) \qquad (6\text{-}13)$$

where S_{TE} = effective size growth.

6.6 Size risk

The size values used in the software cost and schedule estimates are derived from a three-step process. The process is defined as follows:

1. Determine the baseline effective size value and the baseline size standard deviation. The effective size estimating method is described in Section 6.4. The nominal size value is the mean value of the skewed size distribution in Figure 6-8. The "Baseline" size value (10 percent probability) is equal to the mean size value minus the mean size standard deviation. The baseline size is generally computed directly from the CARD information.

2. Estimate the maximum size growth (90 percent probability) that is approximated by "Max" in Figure 6-8. The maximum growth value is determined from the developer capability, the maturity of the software product, the product complexity, and the reuse level of pre-existing source code.

3. Determine the mean growth estimate that corresponds to the most likely size value in the cost and schedule estimates. The baseline, mean and max size values are also used in the software risk analysis. The growth estimating approach is described in greater detail in Section 6.6.1.

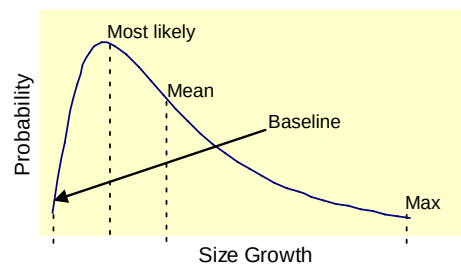


Figure 6-8: Effective size growth distribution

6.6.1 Source code growth

Source code growth is a significant factor in cost and schedule overruns in software development projects. As introduced in Section 6.5.1, Holchin's[32] source code growth model bases physical code growth on software product complexity, product maturity, and software reuse level. The Holchin model growth predictions are comparable to qualitative historical data. The model described in this guidebook to project the effective code growth necessary for a realistic development cost and schedule estimate is an extension of the Holchin algorithm.

The Sage model uses the Holchin algorithm extension to project effective code growth

The most optimistic software development cost and schedule estimates are based on the effective software size discussed in Section 6.4. The size growth S_{max} predicted by the extended model corresponds to the maximum size at 90 percent probability; that is, the maximum size has a 90 percent probability of being less than or equal to the maximum value. The corresponding development cost and schedule represents a risk estimate

[32] Holchin, B. Code Growth Study. Dec. 1991.

assuming size growth only. The maximum development impact corresponds to the maximum baseline size estimate with maximum growth.

For practical purposes, we can view the skewed distribution in Figure 6-8 as a triangular size growth distribution to simplify the mean growth calculation. The error introduced by this assumption is negligible. The mean growth size for the triangular distribution is given approximately by the equation:

$$S_{mean} = 0.707S_{baseline} + 0.293S_{maximum} \qquad (6\text{-}14)$$

From the Holchin model, we know that the amount of software growth is also a function of the software product maturity factor. The growth factor decreases as the project maturity increases, as one would expect. Project maturity defines the level of knowledge the developer has about the systems being developed.

6.7 Function points

FP analysis[33] is a method for predicting the *total size* of a software system. FPs measure software size by quantifying the system functionality provided to the estimator based primarily on the system's logical design.

> *Function points measure the size of what the software does, rather than how it is developed and implemented.*
>
> Carol A. Dekkers, 1999

We need to be careful here to point out that there are many flavors of FP counting rules. Classic FPs in this guidebook conform to the definition in the Function Point Counting Practices Manual, Release 4.2. The ISO FP rules are derived from Release 4.1 of the FP manual. 3D FPs are an extension of the International FP Users Group (IFPUG) Release 4.1 developed by Steve Whitmire[34] of the Boeing Airplane Company for the 7X7 aircraft series. Feature Points were created by Capers Jones[35] of Software Productivity Research for real-time and systems software. There are other FP variations almost too numerous to mention. This guidebook uses the IFPUG 4.2 rules and the 3D extension to briefly describe the use of FPs in software size estimation.

6.7.1 Function point counting

The overall objective of the FP sizing process is to determine an adjusted function point (AFP) count that represents the functional size of the software system. There are several steps necessary to achieve this goal. The procedure is as follows:

1. Determine the application boundary.
2. Identify and rate transactional function types to determine their contribution to the unadjusted function point (UFP) count.
3. Identify and rate data function types to determine their contribution to the UFP count.
4. Determine the value adjustment factor (VAF). This factor adjusts the total size to satisfy operational requirements.

---

[33] International Function Point Users Guide. Function Point Counting Practices Manual: Release 4.2. IFPUG. Westerville, OH: 2004. (Release 4.2 makes it possible to correctly separate size and effort in an estimate.)

[34] Whitmire, S. 3D Function Points: Scientific and Real-Time Extensions to Function Points. Proc. of the 10th Annual Pacific Northwest Software Quality Conference, 1992.

[35] Jones, Capers. Applied Software Measurement: Assuring Productivity and Quality. McGraw-Hill. New York NY: 1997.

5. Calculate the AFP count.

A couple of basic terms, *application boundary* and *transaction*, apply to FP counting.

The application boundary is an imaginary line that separates the application from its external environment. Identification of component types for all dimensions depends on the proper placement of this line.

A transaction is a group of data and operations defined by the application domain. Transactions must have the property of crossing the application boundary. Data in a transaction can flow either in one direction or both ways across the application boundary. A unique transaction is identified by a unique set of data contents, a unique source and/or destination, or a unique set of operations.

Since the rating of transactions and data functions is dependent on both information contained in the transactions and the number of files referenced, it is recommended that transactions be counted first.

The UFP count is determined in steps 2 and 3 (which are discussed later in this section); it is not important if step 2 or 3 is completed first. In graphical user interface (GUI) and object-oriented (OO) type applications, it is easier to begin with step 2.

The AFP count is a combination of both the UFP count and the general system characteristics (GSC) adjustment which is discussed in Section 6.7.4. The AFP count is the adjusted total system size.

6.7.2 Function point components

This section introduces and describes the components utilized in FP analysis. There are seven components:

1. Internal logical files (ILF)
2. External interface files (EIF)
3. External inputs (EI)
4. External outputs (EO)
5. External inquiries (EQ)
6. Transforms (TF)
7. Transitions (TR)

The last two components (transforms and transitions) are used only by the 3D FP-counting approach. 3D FPs are built upon a 3D model of software size. The model was the result of research into the sizing concerns with development of database, scientific, and real-time applications.

Tom DeMarco was the first to characterize software as having three dimensions:[36] data, function, and control. The size contribution of the function dimension is captured by the transform FP component. For example, a weather prediction function has few inputs and outputs, small logical file requirements, and considerable mathematical computations (not adequately dealt with in traditional FP analysis).

The control dimension is concerned with the state-specific behavior of the software system and includes issues such as device control, events, responses, and timeliness. This leads to the need to consider the size of the

[36] DeMarco, Tom. Controlling Software Projects. Yourdon Press, 1982.

behavior-related (states and transitions) aspects of the system. Examples of control dimension software are avionics and process control, neither of which is adequately covered in the classic FP (data) dimension.

The FP approach described in this guidebook accurately represents the IFPUG 4.2 counting definition by ignoring the transform and transition components. However, ignoring them will result in low FP counts for many applications covered by this guidebook.

In addition, we must consider the application boundary, which is not a component, but an imaginary line that separates the elements to be developed, modified, and/or enhanced from the system elements that reside outside the project scope.

There are some definitions necessary to help in rating the first five FP components:

1. Record Element Type (RET)
2. File Type Referenced (FTR)
3. Data Element Type (DET)
4. Processing Steps (PS)
5. Semantic Statement (SS)

All of the classic FP components are rated based upon RETs, FTRs and DETs. As shown in Table 6-5, the transform component of the 3D FPs is rated according to the number of PSs and SSs. The rating measures are described in detail in the sections associated with the component types.

Table 6-5: Function point rating elements

Component	RETs	FTRs	DETs	PSs	SSs
Internal logical files	•		•		
External interface files	•		•		
External inputs		•	•		
External outputs		•	•		
External inquiries		•	•		
Transforms				•	•
Transitions					

6.7.2.1 Application boundary

Computer software systems typically interact with other computer systems and/or humans. We need to establish a boundary around the system to be measured (sized) prior to classifying components. This boundary must be drawn according to the sophisticated user's point of view, as shown in Figure 6-9. Once the border has been established, components can be classified, ranked, and tallied. In short, if the data is not under control of the application or the development, it is outside the application boundary. For business and scientific applications, this boundary is often the human-computer interface. For real-time applications, the boundary can also be the interface between the software and those external devices it must monitor and control.

The internal logical file (ILF) is the data contained entirely inside the system boundary and maintained through the application transactions. ILFs define the internal data structure of the system to be developed. Data maintenance includes the addition, deletion, modification, or access by a transaction.

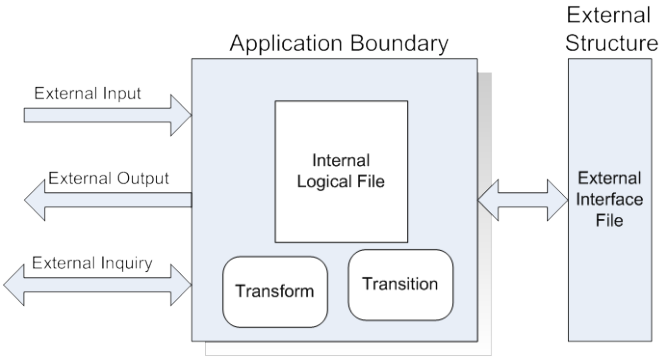


Figure 6-9: Function point system structure

In order to identify the application boundary, consider:

- Reviewing the purpose of the FP count
- Looking at how and which applications maintain data
- Identifying the business areas that support the applications

An external interface is data that is maintained by another application that is directly accessed by our software system without using the services of the external system. Our software system may not modify the external data. If the data is modified, the data is considered to be within the application boundary and part of our system.

Under OO design and practices, external interfaces should not be used. It is frowned upon to directly access data maintained by another application as it violates the principles of encapsulation and data hiding emphasized by software development best practices.

The boundary may need to be adjusted once components have been identified. In practice, the boundary may need to be revisited as the overall application is better understood. Also, FP counts may need to be adjusted as you learn more about the application.

6.7.2.2 Internal Logical File

An ILF is a user-identifiable group of logically related data that resides entirely within the application boundary and is maintained through external inputs (EI); maintaining is the process of modifying data (adding, changing, and deleting) via an elementary process (namely an EI).

Even though it is not a rule, an ILF should have at least one external output and/or external inquiry. That is, at least one external output and/or external inquiry should include the ILF as an FTR. Simply put, information is stored in an ILF, so it can be used later. The external outputs (EO) or external inquiries (EQ) could be to/from another application.

Again, even though it is not a rule, an ILF should also have at least one external input. If an ILF does not have an EI, then one can ask where the information for the ILF comes from. The information must come from an EI. Understanding this relationship improves the thoroughness of the FP count.

A ***DET*** is a unique user-recognizable, non-recursive (non-repetitive) field. A DET is information that is dynamic and not static. A dynamic field is read from a file or created from DETs contained in an FTR. Additionally, a DET can invoke transactions or be additional information regarding transactions. If a DET is recursive then only the first occurrence of the DET is considered, not every occurrence.

An ***RET*** is a set of DETs that are treated by the software system as a group. RETs are one of the most difficult concepts in FP analysis to understand. In a normalized data or class model, each data entity or class model will generally be a separate internal data structure with a single record type. In a parent-child relationship, there is a one-to-one association.

Most RETs are dependent on a parent-child relationship. In this case, the child information is a subset of the parent information; or in other words, there is a one-to-many relationship. For example, consider a warehouse stock display program. One logical group may contain the stock inventory and a second logical group contains a detailed description of the stock items.

One configuration is linked by a key (stock item number) to the data contained in the two files. Thus, the FP count produces 2 ILFs with 1 RET each.

In an aggregation, or collection, data relationships exist between two or more entities. Each entity is a record type and the entire relationship is considered a single internal logical file. The aggregate structure requires a boundary around both groups. Each group is considered to be a record type with a common internal data structure. The aggregate structure yields a FP count of 1 ILF and 2 RETs.

ILFs can contain business data, control data, and rules-based data. It is common for control data to have only one occurrence within an ILF. The type of data contained in an ILF is the same type of data that is possible for an EI to contain.

There are some special cases that need to be explained, *Real Time and Embedded Systems* and *Business Applications*. Real Time and Embedded Systems, for example, Telephone Switching, include all three file types: Business Data, Rule Data, and Control Data. Business Data is the actual call, Rule Data is how the call should be routed through the network, and Control Data is how the switches communicate with each other. Like control files, it is common that real-time systems will have only one occurrence in an internal logical file.

Another case includes *Business Applications*. Examples of business data are customer names, addresses, phone numbers, and so on. An example of Rules Data is a table entry that tells how many days customer payments can be late before they are turned over for collection.

Table 6-6: Table of weights for function point calculations

Component Types	Ranking		
	Low	Average	High
Internal Logical File	7	10	**15**
External Interface File	5	7	**10**
External Input	3	4	**6**
External Output	4	5	**7**
External Inquiry	3	4	**6**
Transform	7	10	**15**
Transition		3	

Ranking:

Like all components, ILFs are rated and valued. The rating is based upon the number of data elements (DETs) and the record types (RETs). Table 6-6 lists the level (low, average, or high) that will determine the number of unadjusted FPs in a later step.

Counting Tips:

Determine the appropriate number of RETs first (Table 6-7). If all or many of the files contain only one record type, then all that is needed to be known is if the file contains more or less than 50 DET's. If the file contains more than 50 data elements, the file will be rated as average. If the file contains less than 50 data element types, the file will be considered low. Any files that contain more than one record type can be singled out and counted separately.

Table 6-7: Ranking for Internal Logical and External Interface Files

	1-19 DETs	20-50 DETs	51+ DETs
1 RET	Low	Low	Average
2-5 RETs	Low	Average	**High**
6+ RETs	Average	**High**	**High**

The relative size impact of each of the FP component types is summarized in Table 6-6. If the component evaluation results in a low ranking, the total number of FPs is multiplied by the weight in the first ranking column. For example, an evaluation produces 3 internal logical files for the application. The number of unadjusted FPs corresponding to 3 low ILFs is 21 (3 x 7) as shown in Table 6-8.

Table 6-8: Unadjusted function point calculation

Element Types	Ranking			Total
	Low	Average	High	
Internal Logical File	3 x 7 = 21	5 x10 = 50	1 x 15 = 15	86

Assume our ILF count process produces 3 low complexity ILFs, 5 average size ILFs and 1 high-complexity ILF. The total ILF function points for the application will be 86 UFPs, as depicted in Table 6-8.

6.7.2.3 External Interface File

An EIF is a user-identifiable group of logically related data that resides entirely outside the application and is maintained by another application. The EIF is an ILF for another application, primarily used for reference purposes.

Maintained refers to the fact that the data is modified through the elementary process of another application. Values are updated as necessary. An *elementary process* is the smallest unit of activity that has meaning to the user. For example, a display of warehouse stock may be decomposed into several subprocesses such that one file is read to determine the stock on hand and another file is read to obtain a description of the stock. The issue is the elementary process.

6.7.2.4 External Input

An EI is an elementary process that handles data or control information coming from outside the application boundary. This data may come from a data input screen, electronically (file, device, interface, etc.), or from another application. The primary intent of an EI is to maintain one or more ILFs and/or to alter the behavior of the system. Calculated and stored values are external input data elements. Calculated values that are not stored are not external input data elements. Calculated values that are not stored have not crossed the application boundary and do not maintain an ILF. Elements consist of data or control information. The data, or control information, is used to maintain one or more internal logical files. Control information does not have to update an ILF.

It is common for information in GUI or OO environments to move from one display window to another. The actual movement of data is not considered an external input because the data has not crossed the application boundary and does not maintain an ILF.

An ***FTR*** is a file type referenced by a transaction. An FTR must also be an ILF or an EIF.

The external input ranking is defined in Table 6-9. The total number of UFPs attributable to external inputs is obtained from Table 6-6 by ranking the EIs, adjusting the total of each ranked group by the EI weighting factors, and accumulating the EI subtotal (as in Table 6-8 for the ILFs).

Table 6-9: Ranking for External Inputs

	1-19 DETs	20-50 DETs	51+ DETs
0-1 FTRs	Low	Low	Average
2-3 FTRs	Low	Low	High
4+ FTRs	Average	High	High

6.7.2.5 External Output

An EO is an elementary process that sends derived data or control information outside the application boundary. The primary intent of an external output is to present information (reports) to a user or output files to

other applications. An external output may also maintain one or more ILFs and/or alter the behavior of the system. Since the information was inside the boundary, it must be contained in an ILF or EIF.

Derived data is data that is processed beyond direct retrieval and editing of information from internal logical files or external interface files. Derived data is the result of algorithms and/or calculations.

The external output ranking is defined in Table 6-10. The total number of UFPs attributable to external outputs is obtained from Table 6-6 by ranking the EOs, adjusting the total of each ranked group by the EO weighting factors, and accumulating the EO subtotal (as in Table 6-8 for the ILFs).

Table 6-10: Ranking for External Outputs

	1-5 DETs	6-19 DETs	20+ DETs
0-1 FTRs	Low	Low	Average
2-3 FTRs	Low	Low	High
4+ FTRs	Average	High	High

6.7.2.6 External Inquiry

An EQ is an elementary process with both input and output components that result in data retrieval from one or more internal logical files and external interface files. This information must be sent outside the application boundary. The processing logic contains no mathematical formulas or calculations, and creates no derived data. *No ILF* is maintained during the processing, nor is the behavior of the system altered.

The IFPUG definition explicitly states the information must be sent outside the application boundary (like an EO). The movement outside the application boundary is important in OO applications because objects communicate with each other. Only when an object actually sends something outside the boundary is it considered an external inquiry.

It is common in the GUI and OO environments for an EO to have an input side. The only distinguishing factor is that an EQ cannot contain derived data. This characteristic distinguishes an EQ from an EO.

The external inquiry ranking is defined in Table 6-11. The total number of UFPs attributable to external inquiries is obtained from Table 6-6 by ranking the EQs, adjusting the total of each ranked group by the EQ weighting factors, and accumulating the EQ subtotal as in Table 6-8 for the ILFs.

Table 6-11: Ranking for External Inquiries

	1-19 DETs	20-50 DETs	51+ DETs
1 FTR	Low	Low	Average
2-5 FTRs	Low	Low	High
6+ FTRs	Average	High	High

6.7.2.7 Transforms

TFs involve a series of mathematical calculations that change input data into output data. The fundamental nature of the data can also be changed as a result of the transformation. New data can be created by the transform. In simple terms, large computational elements are counted by transform FPs in scientific and engineering applications.

Calculations in the real world are often constrained by conditions in the problem domain. These conditions may constrain the possible values assumed by the input data or directly constrain the calculation process itself. These conditions take the form of preconditions, invariants, and post conditions attached to the transforms. Alan Davis,[37] requirements management and software development researcher and professor, referred to the conditions as *semantic statements*. The size of the transformation is

[37] Davis, A. Software Requirements Analysis and Specification. Prentice-Hall, Inc. Englewood Cliffs, N.J.: 1990.

determined by the number of *processing steps* in the mathematic language of the problem domain.

TFs are ranked according to the number of processing steps contained in the TF and the number of semantic statements controlling them.

An example of a TF is the set of calculations required to convert satellite position data into velocity and direction data. The processing steps are:

1. Calculate the change in longitude: $\Delta x = x_2 - x_1$
2. Calculate the change in latitude: $\Delta y = y_2 - y_1$
3. Calculate the direction of travel: $\alpha = \tan^{-1} \frac{\Delta x}{\Delta y}$
4. Calculate the change in altitude: $\Delta z = z_2 - z_1$
5. Calculate the distance in 2 dimensions: $d_2 = \sqrt{\Delta x^2 + \Delta y^2}$
6. Calculate the distance in 3 dimensions: $d_3 = \sqrt{d_2^2 + \Delta z^2}$
7. Calculate the elapsed time: $t_e = t_{start} - t_{stop}$
8. Calculate the satellite velocity: $s = d_3 / t_e$

The satellite data transformation is subject to two preconditions:

1. Latitude change: $\Delta y > 0$ and
2. Elapsed time $t_e > 0$.

The resulting TF rating is based on 8 processing steps and 2 semantic statements.

The mathematical TF must not be confused with an algorithm. An algorithm is a means of embedding a transform in an application, but can be used in other ways. Algorithms are a means of implementing decision processes. TFs *always* involve mathematics. Another way of looking at the difference between algorithms and transforms is algorithms are described in terms of steps and decisions; a transform is always described in terms of calculations.

The transform ranking is defined in Table 6-12. The total number of UFPs attributable to transforms is obtained from Table 6-6 by ranking the TFs, adjusting the total of each ranked group by the TF weighting factors, and accumulating the TF subtotal as in Table 6-8 for the ILFs.

Table 6-12: Ranking for Transforms

	1-5 SSs	6-10 SSs	11+ SSs
1-10 PSs	Low	Low	Average
11-20 PSs	Low	Low	High
21+ PSs	Average	High	High

6.7.2.8 Transitions

TRs are generated by external events to which the application must respond. The transition represents the event-response pair in the application model. In other words, the transitions represent the behavior model and the control dimension of the application.

The data dimension (files, inputs, outputs and inquiries) describes the structure of any given state and the domains from which the behavior is specified, but cannot describe the behavior of the application. This

dimension is the domain for all 3D FP elements except for the TR component type.

Events and responses are characteristics of the problem domain and are a dominant part of the system (application) requirements. TRs are driven by the system requirements. A requirements statement such as "When the target is within 30 yards, the system shall..." defines a state TR.

A sample state-transition diagram is shown in Figure 6-10. There are four distinct states and five *unique* transitions between states as seem in Figure 6-10. Each transition contains an event that triggers the transition and a set of responses activated by the event. The responses can be sequential or concurrent. Multiple events triggering the same response set are considered to be a single unique transition.

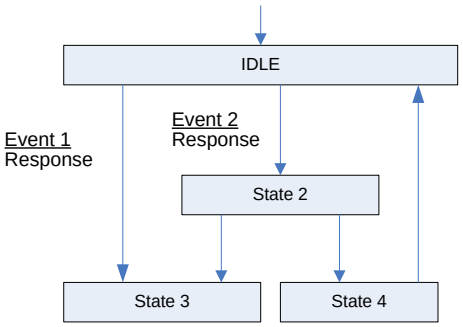


Figure 6-10: State transition model

The transition ranking is defined in Table 6-6. The total number of UFPs attributable to transitions is obtained by multiplying the number of unique transitions by 3 (conversion of transitions to UFPs).

6.7.3 Unadjusted function point counting

The number of instances of a component type for each rank is entered into the appropriate cell in Table 6-13. Each count is multiplied by the weighting factor given to determine the weighted value. The weighted values in each of the component rows are summed in the Total column. The component totals are then summed to arrive at the total UFP count.

Table 6-13: Unadjusted function point calculation

Component Types		Ranking		Total
	Low	Average	High	
Internal Logical File	__ x 7 =	__ x10 =	__ x15 =	
External Interface File	__ x 5 =_	__ x 7 =	__ x10 =	
External Input	__ x 3 =_	__ x 4 =	__ x 6 =	
External Output	__ x 4 =_	__ x5 =	__ x 7 =	
External Inquiry	__ x 3 =_	__ x 4 =	__ x 6 =	
Transform	__ x7 =_	__ x10 =	__ x15 =	
Transition		__ x 3 =		
Unadjusted Function Points				

Remember, the UFP count arrived at in Table 6-13 relates to the total functional size of an average software system with the requirements and structure defined by the FP count. There have been no adjustments for the system type or processing requirements. In a real sense, we cannot distinguish between a data processing system and a real-time avionics system. We cannot discern the difference between a system with a strong user interface requirement and a space payload. What we do know is the system contains a certain average amount of functionality. We have measured *only* the total system size. The UFP number tells us nothing about the development or the operational environment.

6.7.4 Adjusted function points

AFPs account for variations in software size related to atypical performance requirements and/or the operating environment. It is likely that the software system you are measuring will be larger than average when there are requirements for reusability or user support. Consequently, there must be a corresponding adjustment to the total size projection to deal with this type of issue. Predicting total size from a SLOC point of view automatically adjusts the size to deal with atypical conditions. Atypical conditions are not accounted for in the UFP frame of reference. Hence, there is a necessity to adjust the UFP count to compensate for these non-average requirements.

There is a trend in modern software estimating tools to ignore the size adjustment in producing the software development cost and schedule

estimate. The component-level estimating model (Appendix G) contains a set of factors that adjust the cost and schedule estimate to account for development environment variations with atypical requirements and constraints. The primary argument against adjusting the FP size projections is that the cost and schedule impacts are already dealt with in the effort adjustments. Herein lies the faulty assumption: Adjustments are *required* in both size and effort. More code (software) is necessary to implement user support features and more effort is required to design and test those features.

6.7.4.1 Value Adjustment Factor

Table 6-14: General System Characteristics definitions

No.	General Characteristic	Description
1	Data communications	How many communication facilities are there to aid in the transfer or exchange of information with the application or system?
2	Distributed data processing	How are distributed data and processing functions handled?
3	Performance	Did the user require response time or throughput?
4	Heavily used configuration	How heavily used is the current hardware platform where the application will be executed?
5	Transaction rate	How frequently are transactions executed daily, weekly, monthly, etc.?
6	Online data entry	What percentage of the information is entered online?
7	End-user efficiency	Was the application designed for end-user efficiency?
8	Online update	How many internal logical files are updated by online transactions?
9	Complex processing	Does the application have extensive logical or mathematical processing?
10	Reusability	Was the application developed to meet one or many user's needs?
11	Installation ease	How difficult is conversion and installation?
12	Operational ease	How effective and/or automated are start-up, backup, and recovery procedures?
13	Multiple sites	Was the application specifically designed, developed, and supported to be installed at multiple sites for multiple organizations?
14	Facilitate change	Was the application specifically designed, developed, and supported to facilitate change?

The VAF is the mechanism used by the FP methodology to adjust software size projections for special requirements placed upon the software. The adjustment is based on a series of general system characteristics (GSCs) that determine an overall VAF. Combining the UFP total and the VAF produces the AFP value for the system. There are 14 GSCs defined in Table 6-14 that rate the general functionality of the system and produce the software size projection. The 14 GSCs can be evaluated early in the software development cycle since they are generally part of the software requirements.

The impact (degree of influence) of each of the system characteristics is rated on a scale of 0 to 5 in response to the question in the general characteristic description. The detailed definition of each characteristic is based on the IFPUG 4.2 guidelines. The ratings are classified according to the information in Table 6-15.

Table 6-15: General System Characteristic ratings

Rating	Definition (relative impact)
0	Not present, or no influence
1	Incidental influence
2	Moderate influence
3	Average influence
4	Significant influence
5	Strong influence throughout

To demonstrate use of the rating scheme, we have selected the Online Data Entry (GSC 6) characteristic. The corresponding GSC 6 ratings are shown in Table 6-16.

Table 6-16: Online Data Entry rating definitions

Rating	Definition
0	All transactions are processed in batch mode
1	1% to 7% of transactions are interactive data entry.
2	8% to 15% of transactions are interactive data entry
3	16% to 23% of transactions are interactive data entry
4	24% to 30% of transactions are interactive data entry
5	More than 30% of transactions are interactive data entry

GSC items such as transaction rates, end user efficiency, on-line update and reusability have higher values for user friendly (GUI) systems than for traditional systems. Conversely, performance, heavily used configurations, and multiple site installations have lower GSC values for user friendly systems than for traditional systems.

Once the 14 GSCs have been evaluated, the value adjustment factor can be computed using the following IFPUG VAF equation:

$$VAF = 0.65 + \left[\left(\sum_{i=1}^{14} GSC_i \right) / 100 \right] \qquad (6\text{-}15)$$

where GSC_i = degree of influence for each General System Characteristic and
i = 1 to 14 representing each GSC.

The resulting GSC score is between 0.65 and 1.35 depending on the sum of the individual ratings.

6.7.4.2 Adjusted function point calculation

The AFP count is obtained by multiplying the VAF by the UFP count. The standard AFP count is given by:

$$AFP = VAF \times UFP \qquad (6\text{-}16)$$

where AFP = Adjusted Function Point count,
VAF = Value Adjustment Factor, and
UFP = Unadjusted Function Point count.

6.7.5 Backfiring

The vast majority of widely used software estimating tools are based on SLOC for predicting development effort and schedule. Function points, albeit a viable total software size estimating approach, are not directly compatible with the algorithms used by these tools. Note that some of these tools do allow the input of function points as a size measure, but internally convert function points to SLOC before the cost and schedule are estimated. Research into conversion factors between function points and SLOC have been conducted by several researchers, the most comprehensive of these studies was published by Capers Jones[38] in 1991. Table 6-17 contains a commonly used subset of the language conversions listed by Capers Jones. The SLOC/FP and range data listed here is a compendium of the data published by the various researchers. The table listing correlates well with the Capers Jones data.

[38] Jones, Capers. Applied Software Measurement. McGraw-Hill Inc. New York, NY: 1991.

Table 6-17: FP to SLOC conversion

Language	SLOC/FP	Range	Language	SLOC/FP	Range
Ada 95	49	40-71	JOVIAL	107	70-165
Assembler, Basic	320	237-575	Object-oriented	29	13-40
Assembler, Macro	213	170-295	Pascal	91	91-160
BASIC, ANSI	32	24-32	PL/I	80	65-95
C	128	60-225	PROLOG	64	35-90
C++	55	29-140	CMS-2	107	70-135
COBOL (ANSI 95)	91	91-175	3rd generation	80	45-125
FORTRAN 95	71	65-150	4th generation	20	10-30
HTML 3.0	15	10-35	Visual Basic 6	32	20-37
JAVA	53	46-80	Visual C++	34	24-40

The data in the table illustrates the relative efficiencies of modern programming languages. For example, the ratio of macro assembler to C++ is 213/55 = 3.87 showing the size advantage of C++ over assembler is nearly 4:1. The ratio suggests a large cost advantage in the use of high-level languages.

Please note the wide range for each of the SLOC/FP ratios in the table. Using the backfiring techniques to obtain equivalent source code size values will not yield precise SLOC counts. **Also remember the backfiring process does not produce the "effective" source code size used in cost and schedule estimates**. The backfire process produces a *total* functional size estimate that assumes no software reuse or COTS use.

The conversion of function points (UFP or AFP) to equivalent total SLOC is accomplished with the equation:

$$S_t = AFP \times BF \qquad (6\text{-}17)$$

where S_t = total software SLOC count,

AFP = Adjusted Function Point count, and

BF = SLOC/FP conversion factor from Table 6-17.

The AFP value yields the most rational conversion to source lines of code.

6.7.6 Function points and objects

3D FP sizing (introduced in Section 6.7.2) and object sizing are essentially identical technologies. At a high level, applications and objects have the same FP component properties. Applications have internal data, contain mathematical transforms, and have externally observable behavior. Applications are made from lower-level objects that can be either concurrent or contained within other objects. To understand this concept, we must map the FP components to the features of a class.

To perform the mapping, establish the boundary of the class as you would a FP application. Anything that crosses the object (class) boundary is an input,

output, or inquiry. A message that passes information into the object and triggers a state change is an input. A message that is produced by the class is an output. A message that causes the object to return information about its state without causing the object to change state is an inquiry.

The object's static internal structure describes the properties of the object and represents the possible states the object can assume. Grady Booch,[39] software designer and methodologist, says, "The state of an object encompasses all of the properties of the object plus the current values of each of the properties." The structure of the object's state space is its internal data structure. As a minimum, an object contains at least one record type, that of the state representation. The properties of the state are the record data element types.

Objects can contain mathematical transformations. Not all objects contain TFs. Some are purely algorithmic in nature. The objects that do contain mathematical TFs are identified as transformers or converters.

Objects generally exhibit an external behavior. The way an object reacts to an external event (message) is dependent upon the contents of the message and the current state of the object. All objects exhibit some sort of behavior. The transitions of the behavior model are the transitions in the FP domain.

3D FPs can be applied directly in OO software where the boundary is drawn around the application level. At this level, 3D FPs measure the function delivered, not necessarily the functionality developed.

6.7.7 Zero function point problem

A new development can be specified using a FP approach and/or a SLOC-based approach. We can also specify the size of new functionality within an existing system in terms of FPs, but we most often use SLOC to measure modification of existing functionality. When the estimate is for extension of an existing system, the size specification generally becomes more complex. Care must be taken to ensure that the portions specified using FPs do not overlap the portion expressed in SLOC and that the system development is completely covered by the composite of the two approaches.

The major issue with using FPs in this situation is called "the zero function point problem." When using the FP sizing approach, an issue arises when a portion of the software is being modified without adding to or changing the system functionality. The FP concept allows for specification of the total system development size only. The size of added system functionality is only a problem of defining the application boundary for the development. FPs do not attempt to represent development effort; thus, there is no mechanism for dealing with modifications, deletion, regression testing, and reverse-engineering effort.

The zero FP issue can only be dealt with through the effective SLOC. The effective size can be specified as:

$$S_{eff} = S_{FP} + S_{\text{mod}} + S_{reused} SAF \qquad (6\text{-}18)$$

where S_{FP} is the adjusted FP count converted to SLOC through the backfiring technique and

SAF is the Size Adjustment Factor defined in Equation (6-4).

[39] Booch, G. Object-Oriented Analysis and Design With Applications, Second Edition. Benjamin-Cummings, 1994.

In the SLOC coordinate system, the development effort for the modifications and reused software can be computed. This approach requires the FP approach be limited to the new development size projection and the SLOC-based approach be used for modification and reuse portions of the software product.

There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.

Frederick P. Brooks, Jr.

Section 7

Productivity Factor Evaluation

7.1 Introduction

The system-level estimating process, considered a first-order model, introduced in Section 4 is the simplest of the software estimating models. The model consists of a productivity factor (PF) multiplied by the software product effective size to obtain the development effort. The production units (size) can be effective source lines of code (ESLOC), function points, object points, use cases, and a host of other units as long as the PF units match the production units. The development effort can be a person-hour (PH), person-month (PM), or any other unit corresponding to the unit measure of the PF. For the purpose of this introductory discussion, we will use effective source lines of code (ESLOC) as the production unit measure and PH per effective source line of code as the productivity measure. This can be stated as:

$$E_d = C_k S_e \qquad (7\text{-}1)$$

where E_d is the development effort in PH,

C_k is a productivity factor (PH/ESLOC), and

S_e is the number of ESLOC.

The PF is commonly determined by the product type, historic developer capability, or both, derived from past development projects. As simple as this equation is, it is widely used to produce high-level, rough estimates.

As an example, the development effort for a military space payload with S_e = *72* KESLOC, and a PF of 55 lines per person-month (ESLOC/PM). Note that a month is assumed to be 152 hours by most estimating tools. The hours per ESLOC value is simply:

*PF = 55 ESLOC/PM * (PM/152 PH)*
= 0.362 ESLOC/PH
= 2.76 PH/ESLOC

E_d *= 2.76 PH/ESLOC * 72000 ESLOC*
= 198,720 PH

*198,720 PH * PM/152 PH = 1,307 PM*

or

$$2.76\frac{PH}{ESLOC} = \left(152\frac{PH}{PM} \div 55\frac{ESLOC}{PM}\right) \qquad (7\text{-}2)$$

The development effort for this project is then 198,720 PH or 1,307 PM.

The utility of the first-order model is that it can be used early in the development process (typically before Milestone A) before the software system architecture is established and the software components defined. The model is also valuable where there is little or no knowledge available of the developer or the developer's environment. The first-order model assumes a

generic, or average, software development organization and no special constraints on the software product. A reasonable ballpark estimate can be created quickly to support rough cost planning.

The C_k value depends on the product type. The product type is generally obtained from historical data or from productivity tables. Product types are divided into 13 categories in the guidebook productivity tables to provide meaningful PFs for these types of systems. The product characteristics for each class are typical for the product category. Additional system types can be covered by selecting the closest approximation to the given product type.

One weakness of the first-order model is its insensitivity to the magnitude of the effective product size. Productivity is, or at least should be, decreased for larger projects. The tables included in this section partially compensate for the system size issue. Interpolation between specific size values in the tables refines the PF values.

Once a reasonable PF value has been selected from the table, it is a simple matter to multiply the PF by the estimated effective size to obtain a total cost or effort estimate for the project. The tables in this section assume an average development organization operating in a normal environment. Subjective adjustments can be made to the factors to compensate for application experience (and so forth) if supported by historic project data.

7.2 Determining productivity factor

Typically, for estimating purposes, a PF is chosen based on historical data from similar past development efforts. The historic factor is calculated by dividing the total development effort (in PM) by the total effective size of the software, in ESLOC or thousand source lines of code (KESLOC), to obtain the PF (PM/ESLOC). Inverting the PF yields the familiar ESLOC/PM productivity metric. It is worth stressing that these productivity values include all software development activities from design through software component integration.

By collecting several data points from a development organization for a specific product type, an average PF can be computed that will be superior to the factors tabulated in this section. Collecting several data points of different project sizes will produce a size-sensitive productivity relationship. Table 7-1 shows some typical productivity values for various software application types.

The productivity figures in Table 7-1 can be converted from effective source lines per person month to person months per thousand lines (KESLOC) to satisfy the first-order model calculations in Equation (7-1). The results of the conversion are contained in Table 7-2 repeated here for convenience. A similar table extracted from the QSM database by McConnell is contained in McConnell's *Software Estimation*[40].

[40] McConnell, S. Software Estimation. Microsoft Press Redmond, WA: 2006.

Table 7-1: Typical productivity factors (ESLOC/PM) by size and software type and stated complexity (D) value

Software Type	D	10 KESLOC	20 KESLOC	50 KESLOC	100 KESLOC	250 KESLOC
Avionics	8	79	69	57	50	42
Business	15	475	414	345	300	250
Command and Control	10	158	138	115	100	83
Embedded	8	111	97	80	70	58
Internet (public)	12	475	414	345	300	250
Internet (internal)	15	951	828	689	600	500
Microcode	4-8	79	69	57	50	42
Process Control	12	238	207	172	150	125
Real-time	8	79	69	57	50	42
Scientific Systems/ Engineering Research	12	396	345	287	250	208
Shrink wrapped/ Packaged	12-15	475	414	345	300	250
Systems/ Drivers	10	158	138	115	100	83
Telecommunication	10	158	138	115	100	83

Source: Adapted and extended from McConnell, *Software Estimation*, 2006, Putnam and Meyers, *Measures for Excellence*, 1992, Putnam and Meyers, *Industrial Strength Software*, 1997 and Putnam and Meyers, *Five Core*.

Table 7-2: Typical productivity factors (PM/ KESLOC) by size and software type

Software Type	D	10 KESLOC	20 KESLOC	50 KESLOC	100 KESLOC	250 KESLOC
Avionics	8	12.6	14.5	17.4	20.0	24.0
Business	15	2.1	2.4	2.9	3.3	4.0
Command and Control	10	6.3	7.2	8.7	10.0	12.0
Embedded	8	9.0	10.4	12.4	14.3	17.2
Internet (public)	12	2.1	2.4	2.9	3.3	4.0
Internet (internal)	15	1.1	1.2	1.5	1.7	2.0
Microcode	4-8	12.6	14.5	17.4	20.0	24.0
Process Control	12	4.2	4.8	5.8	6.7	8.0
Real-time	8	12.6	14.5	17.4	20.0	24.0
Scientific Systems/ Engineering Research	12	2.5	2.9	3.5	4.0	4.8
Shrink-wrapped/ Packaged	12-15	2.1	2.4	2.9	3.3	4.0
Systems/ Drivers	10	6.3	7.2	8.7	10.0	12.0
Telecommunication	10	6.3	7.2	8.7	10.0	12.0

7.2.1 ESC metrics

The USAF Electronic Systems Center (ESC) compiled a database[41] of military projects developed during the years 1970-1993 containing 29 development projects. The projects included development data from 91 Computer Software Configuration Items (CSCIs) divided into three categories based on software reliability requirements. The projects and CSCIs are not grouped in the same way as the Software Type categories in Tables 7-1 and 7-2. Care should be taken when using the ESC data outside the domains represented in the ESC database.

The ESC categorization follows the reliability definitions posed by Boehm in *Software Engineering Economics*; that is, grouping according to the following reliability categories[42]:

Very low	The effect of a software failure is simply the inconvenience incumbent on the developers to fix the fault. Typical examples are a demonstration prototype of a voice typewriter or an early feasibility-phase software simulation model.
Low	The effect of a software failure is a low level, easily recoverable loss to users. Typical examples are a long-range planning model or a climate forecasting model.
Nominal	The effect of a software failure is a moderate loss to users, but a situation from which one can recover without extreme penalty. Typical examples are management information systems or inventory control systems.
High	The effect of a software failure can be a major financial loss or a massive human inconvenience. Typical examples are banking systems and electric power distribution systems.
Very high	The effect of a software failure can be the loss of human life. Examples are military command and control systems, avionics, or nuclear reactor control systems.

Each project contains CSCIs with different ratings. Each CSCI within a project is required to interface with one or more additional CSCIs as part of a composite system. The interaction includes both internal interfaces as well as interfaces to external systems. The number of integrating CSCIs is defined as the total number of CSCIs in the project. ESC formed three categories based on the number of integrating CSCIs and the required reliability level for their productivity analysis, as shown in Table 7-3.

[41] AFMC Electronic Systems Center. "Cost Analysis: Software Factors and Estimating Relationships." ESCP 173-2B. Hanscom AFB, MA: 1994.
[42] Boehm. B.W. Software Engineering Economics.

Table 7-3: Definition of complexity/reliability categories

	Integrating CSCIs		
Reliability	**0- 6 CSCIs**	**7-10 CSCIs**	**> 10 CSCIs**
Very high (Public safety required)	Category 2	Category 3	Category 3
High (Major financial loss)	Category 2	Category 2	Category 3
Very low - Nominal (Moderate loss)	Category 1	Category 1	No Data

Productivity and the associated factors for the three project categories are consistent with the information in the PF tables. An interesting note on the ESC productivity analysis shown in Table 7-4 is the narrow range of the productivity data for each category. Standard deviations for the three categories—even ignoring the project size and development environment—are very low.

Table 7-4: Productivity for military applications by category

Project Type	Productivity (ESLOC/PM)	Productivity Factor (PM/KESLOC)	Productivity Range (ESLOC/PM)	Standard Deviation (ESLOC/PM)
All programs	131.7	7.60		
Category 1	195.7	5.10	116.9 – 260.8	49
Category 2	123.8	8.08	88 – 165.6	23.6
Category 3	69.1	14.47	40.6 – 95.2	16.5

The 91 CSCIs contained in the ESC data, analyzed in terms of development personnel, show the impact of development capability on productivity. The results of the productivity analysis are shown in Table 7-5. The results do not exactly match those given in Table 7-4 because the project categories contain a mix of CSCI categories; that is, a Category 3 project may contain Category 1, 2, and 3 CSCIs. In any case, the productivity variation across the personnel capability range is significant and should be considered when selecting a PF for an estimate. Note the information contained in Tables 7-1 and 7-2 include personnel of all capability levels.

Table 7-5: Productivity for military applications by category as a function of personnel capability

Personnel	Category 1 (ESLOC/PM)	Category 2 (ESLOC/PM)	Category 3 (ESLOC/PM)
Above average	265.2	165.5	91.0
Average	177.9	141.6	45.4
Below average	No data	101.2	41.5
Total	216.1	134.3	53.9

7.2.2 Productivity index

The Quantitative Software Management (QSM) Software Lifecycle Model (SLIM®) introduced in Section 4 is the prime example of use of a PF and of a second-order estimating model. The general form of the SLIM® software equation is:

$$S_e = C_k \sqrt[3]{K} T^{4/3} \qquad (7\text{-}3)$$

where K = the total life cycle cost (person-years),

C_k = a PF relating cost and schedule to the effective size, and

T = the full-scale development schedule (years)

A second commonly used measure of developer capability is the Putnam Productivity Index (PI)[43,44] implemented in the SLIM® Estimating Suite. The Putnam software equation defined in Equation (7-3) uses a software PF C_k to equate the development cost and schedule to the effective software size. The Putnam PF is conceptually equivalent to the effective technology constant defined in Table 7-6. The SLIM® C_k range is from approximately 750 to 3,500,000. To simplify the use of the PF a new variable known as the *PI* was created. The PF range, indicated as PI, is from 0 to 40. The relationship between C_k and *PI* is approximately:

$$C_k \cong 600.7(1.272)^{PI} \qquad (7\text{-}4)$$

The relationship between the productivity index and the PF can easily be visualized from the data in Table 7-6.

Table 7-6: Relationship between C_k and PI values

PI	C_k	PI	C_k
1	754	11	8,362
2	987	12	10,946
3	1,220	13	13,530
4	1,597	14	17,711
5	1,974	15	21,892
6	2,584	16	28,657
7	3,194	17	35,422
8	4,181	18	46,368
9	5,186	19	57,314
10	6,765	20	75,025

The *PI* value contains the impacts of product characteristics and the development environment, as well as the domain experience and capability rating for the organization much like the effective technology constant discussed in Section 5.4.5. *PI* also subsumes the application complexity impact and the effects of software reuse; that is, the impacts of reverse engineering and regression testing are contained in the *PI* value.

Since the *PI* value is conceptually closer to the effective technology constant, the typical range of *PI* values must be specified for each domain or application type, as shown in Table 7-7. Note that a positive PI change of three points represents a doubling of the process productivity.

The PF C_k is approximately double the effective technology constant (Section 5.4.5) C_{te} value for realistic C_k values. The estimating model in Section 5 includes the effects of development environment on the development effort (see Equation [5-1]). Assuming a product environment penalty of 2 units (that is, $\prod_i f_i = 2$), the basic technology constant C_{tb} defined in Section 5.4.4 is roughly equal to C_k. The practical upper basic technology constant limit of 20,000 places *PI* values of 13 or greater at risk in terms of achievable productivity. For example, a software development of 50,000 source lines with a *PI* value of 14 yields a predicted full-scale development productivity of 750 lines per person month.

---

[43] Putnam, L.H., and W. Myers. Measures of Excellence. Prentice-Hall, Inc. Englewood Cliffs, NJ: 1992.

[44] Putnam, L.H., and W. Myers. Five Core Metrics. Dorset House Publishing. New York, NY: 2003.

Productivity index data corresponding to the PI values in Table 7-7 are strongly skewed toward the Low PI value. For example, projects in the business domain typically have a productivity value on the order of 300-400 lines per PM. The Low PI value corresponds to a productivity of about 400 lines per month. The High PI value corresponds to a productivity of approximately 6,500 lines per month. Extrapolating, a 50,000 source line project could be completed in only 8 PM, according to Table 7-7.

Table 7-7: Typical PI ranges for major application types from the QSM database

Domain	Low PI	High PI
Business	12	21
Scientific	10	20.5
System	4	15
Process control	10	17.5
Telecommunications	8	14
Command and Control	7.5	14
Real time	5	13
Avionics	3.5	14.5
Microcode	3.2	9.2

There is an important qualifier for the upper reaches of the Putnam productivity index that is important. Most software development projects consist of CSCIs of 30,000 ESLOC or more. A small project ranges from 5,000 to 15,000 effective source lines; the achievable productivity can reach significant values that cannot be attained in larger projects. In this special case, PI values of 11 or 12 are reasonable. The reason for this extension is the small size of the development team that can work closely to produce the product. Products with significant performance requirements are not likely to reach these values.

The productivity rates presented in Table 7-1 are typical for common project types. A complexity (*D*) value column was added to the table to show the relationship between the software type and complexity. Complexity is defined and discussed in Section 10.1. It is interesting to note the productivity rate achieved for each of the software types tends to group around the associated complexity value. Less complex product types (higher *D* value) have higher productivity for each project size group.

The project size category also tells us about the size and character of the project team. A 10 KESLOC project will likely be developed by a small team, probably a total of 10 or fewer members (including test and integration personnel). The system defined by the project will only consist of one or two CSCIs. The development team (programmers, integrators) will be about five people in a normal environment. Projects of this size are at the lower limits of the data defining the widely used software cost estimating tools. Productivity is very high because of the ability of the development team to communicate and work together.

7.3 System-level estimating

System-level cost estimates, using either the PF model or the ESC model, can quickly produce credible ballpark estimates. Note: *quickly* does not equate to *simply*. The key to system-level estimating is the realism of the effective size projection, which is not always a simple task. Take the following as an example:

> Given an avionic system upgrade consisting of a 1,000 ESLOC modification and an 834 function point (adjusted) addition. The implementing programming language is C++. The avionic system required reliability is at the "loss of human life" level. The required effort for the upgrade using Equation (7-1) will be:
>
> $$E_d = 14.47 * (1{,}000 + 55 * 834) = 678.2 \text{ PM} \qquad (7\text{-}5)$$
>
> according to the ESC model tabulated in Table 7-4. The 46,870 effective size is close to the 50,000 source line column in Table 7-2. The avionics system PF of 11.76 for the extended McConnell approach

yields an effort of 551.2 PM, which is lower than (but reasonably close to) the ESC model results.

There are NO useful methods to project a development schedule at the system level unless the system can be developed as a single CSCI. If the system can be built as a single CSCI, the schedule can be approximated by an equation of the form:

$$T_d \cong 3.5 * \sqrt[3]{E_d} \text{ months} \qquad (7\text{-}6)$$

where T_d is the development time (months), and

E_d is the development effort (PM).

Remember, the results are only as accurate as the estimating model which, in turn, is only as accurate as the underlying data. The estimates are totally independent of the development environment since we have no knowledge of the product characteristics or of the developer contracted to build the system. We are also working with an estimated size which may (probably will) grow between this estimate and the final product.

What we can or cannot do, what we consider possible or impossible, is rarely a function of our true capability. It is more likely a function of our beliefs about who we are.

Tony Robbins

Section 8

Evaluating Developer Capability

One of the most important steps in developing a software cost and/or schedule estimate is the establishment of the software developer's capability rating. The value of this rating changes slowly over time, if it changes at all. There are two primary variables in the capability rating: (1) the capability of the analysts and programmers, and (2) the level of experience in the application domain of the project being estimated.

The first variable that has the greatest impact and changes very slowly is the capability of the analysts and programmers, while the second variable, domain experience, is more likely to change over time and defines the organization's principle product lines.

The developer, in the context of this guidebook, is the development organization, which is made up of analysts (software engineers), programmers, and managers. The developer resides in physical facilities and uses development resources that relate to the developer's normal way of doing business. The developer's normal way of doing business can be visualized as the developer's culture. Culture implies a level of usage that transcends experience, or the developer would not think of using any other tool or practice.

When evaluating an estimate, the first question one asks is "What productivity is this organization capable of achieving?" The developer capability rating defines organization productivity without the load imposed by the project or product constraints.

8.1 Importance of developer capability

Developer capability measures the efficiency and quality of the development process and is one of the two most significant cost and schedule drivers. The second and most important driver is the effective size of the system described in Section 6.

The system-level estimating model described in Section 4 contains a constant productivity factor C_k (person hours per equivalent source line of code) that assumes an average capability and organization efficiency as well as an environment and product characteristics consistent with the application type. The only way we can include organization efficiency is by deriving the productivity factor from historic data collected for a specific organization and product type.

The component-level estimating model extends the productivity factor to include the impacts of the developer and development environment on organization efficiency. This model provides the only mechanism to account for the significant cost drivers: people, environment and product type.

The inherent developer capability is the most significant factor that determines an organization's productivity. The major attributes of a developer capability rating are:

- Problem solving or engineering ability
- Efficiency and thoroughness

- Ability to communicate and cooperate

8.2 Basic technology constant

There are a total of about 35 characteristics that are useful in determining the impact of the product and the development environment on the ultimate project development cost and schedule. The 35 characteristics essentially define the productivity factor. Six of those characteristics are useful in quantitatively determining the "raw" capability of a software development organization. The characteristics are:

- Analyst, or software engineer, capability (ACAP)
- Programmer, or coder, capability (PCAP)
- Application experience (AEXP)
- Use of modern practices (MODP)
- Use of modern development tools (TOOL)
- Hardcopy turnaround time (TURN)

The term *raw capability* isolates the undiluted developer performance from the effects of the product and the environment that reduces the overall project productivity. For example, separating the development team across multiple development sites reduces productivity due to communication difficulties across the sites. The basic technology measure eliminates the degradation by focusing on an ideal set of project conditions.

The basic technology constant (C_{tb}) is a measure of organizational capability in a specific application area. The measure was first implemented in the Seer estimating model in the late 1970s. Since that time, the measure has achieved widespread acceptance and is used in several software estimating tools. An organization's basic technology constant is relatively stable (steady) over time, making it useful as an organization capability measure. The constant is domain (product)-specific.

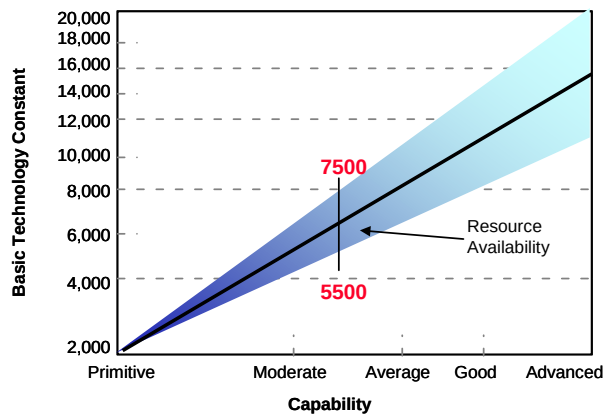


Figure 8-1: Basic technology constant range

The range of values for the measure is from 2,000 to 20,000, as shown in Figure 8-1. An organization with the 2,000 rating is incompetent, unmotivated, has a Capability Maturity Model Integration (CMMI) Level 1 process rating, uses software development tools from the 1960s, and operates on a remote computing system. The organization has no interactive communication capability. This is a true "Dark Age" development organization. As bleak as Dark Age sounds, there are still organizations with basic technology ratings in the low 2,200s.

The high end of the C_{tb} range is a "projected" value of 20,000. Such an organization would be highly motivated and working in a team environment with free, direct communication between team members. The organization would be a CMMI Level 5 entity; that is, one that is fully involved with the principles and philosophy of continuous process improvement. The organization would use modern management and development tools and practices as a matter of culture. Computing resources are instantaneous and within easy reach.

The typical basic technology constant range today is between 5,500 and 7,500 as shown in Figure 8-2. There are a few organizations with basic

technology ratings in the 8,000 to 9,000 range. The significance of the ratings above 8,000 is these organizations are led by managers who practice Theory Y management[45]. The highest data point was achieved in a variation of the Skunk Works©[46] environment (detailed in Section 8.3.3.1).

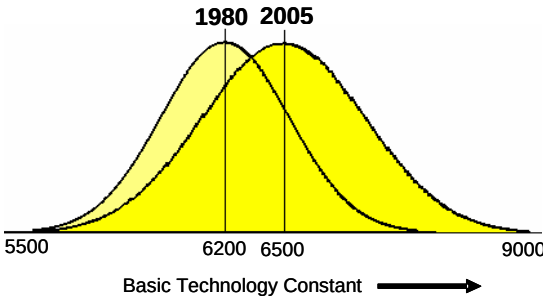


Figure 8-2: Basic technology constant distribution 2005

The mean value of the basic technology constant distribution has changed little over the last 25 years, in spite of dramatic changes to the software development technology. From a productivity point of view, the shift of less than 15 points per year is discouraging as can be seen in Figure 8-2. From an estimating point of view, this is good because the project data that underlies almost all software estimating tools is still current and the tools are stable. An important observation is "*culture is constant*." Estimating templates created for organizations 20 years ago are generally as valid today as they were then.

Another indication of developer capability stability is shown in Figure 8-3. Productivity data from the 1960s through the 1990s indicates a slow but steady growth in productivity of about 1 line per PM per year for Department of Defense (DoD) projects, in spite of the introduction of several major language and technology changes. Each technology listed in Figure 8-3 was introduced with the promise that productivity would increase an order of magnitude and errors would disappear. Some improvements were expected because technology improvements do have a positive effect on productivity.

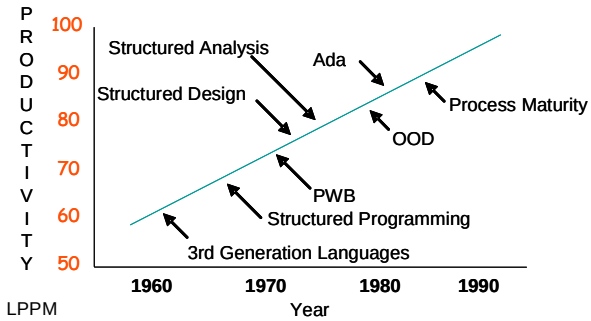


Figure 8-3: Productivity gains from 1960 to present

Since the basic technology constant is relatively stable over a long period of time, it is useful to build a library of technology-constant data for each organization as developing estimates. The process simplifies estimating because the parameters in the basic technology constant are the most difficult to obtain without some calibration using completed projects. The parameter values in the basic constant can be validated through *post-mortem* analysis to increase confidence in the data.

8.2.1 Basic technology constant parameters

8.2.1.1 Analyst capability

The major attributes of the analyst capability rating are:

- Analysis or engineering ability
- Efficiency and thoroughness
- Ability to communicate and cooperate

These attributes are of roughly equal value in the evaluation. The evaluation *should not* consider the level of domain experience of the analysts. The experience effects are covered by other factors. The evaluation should be based on the capability of the analysts as a *team* rather than as individuals. The capability rating can be outlined as:

---

[45] Hersey, P., and K.H. Blanchard. Management of Organizational Behavior, Utilizing Human Resources. Prentice-Hall, Inc. Englewood Cliffs, NJ: 1977.

[46] © 2005, Lockheed Martin Corporation

- Motivation
- Use of team methods
  - Communication ability
  - Cooperation
- Working environment
  - Noise level
  - Individual working (thinking) space
  - Proximity of team members
- Problem-solving skills
- Software engineering ability

This working definition is compatible with the major attributes specified (at the beginning of this section) and focuses on the attributes that have the greatest impact on software cost and schedule.

Motivation, use of team methods, and working environment underscore the importance of management and communication in the capability rating.

The first step of the capability evaluation categorizes the organization's management style as either Theory X or Theory Y[47]. Theory X assumes the organization (or at least the project manager) attitude toward the project personnel can be described as:

1. Work is inherently distasteful to most people.
2. Most people are not ambitious, have little desire for responsibility, and prefer to be directed.
3. Most people have little capacity for creativity in solving organizational problems.
4. Most people must be closely controlled and often coerced to achieve organizational objectives.

> *Never tell people how to do things. Tell them what to do and they will surprise you with their ingenuity.*
>
> George S. Patton, Jr.

Theory Y, which is diametrically opposed to Theory X, assumes:

1. Work is as natural as play, if conditions are favorable.
2. Self-control is often indispensable in achieving organizational goals.
3. The capacity for creativity in solving organizational problems is widely distributed in the population.
4. People can be self-directed and creative at work if properly motivated.

Theory X is often referred to as a *directing* management style, while Theory Y is a *leading* style. The Theory Y management style is easily ascertained by interview with the software development staff. Projects managed by leaders are typified by high motivation and morale. Although most managers claim to practice participative management (leading), numerous research studies indicate that authoritarian (directing) management styles were predominant in software development organizations.

The ACAP rates the personnel who are responsible for the front end or high level design elements of the software development. These people are usually

> *Not every group is a team, and not every team is effective*
>
> Glenn Parker

[47] Hersey, P., and K.H. Blanchard. Management of Organizational Behavior: Utilizing Human Resources. Prentice-Hall, Inc. Englewood Cliffs, NJ: 1977.

classified as software engineers or system analysts. The rating is related to the analyst team performance as shown in Table 8-1. The rating description focuses on motivation and team performance which are the major drivers in the rating.

Table 8-1: Analyst capability ratings

ACAP – Analyst Capability	
Value	Description
0.71	Highly motivated **AND** experienced team organization
0.86	Highly motivated **OR** experienced team organization
1.00	**TRADITIONAL** software organization
1.19	Poorly motivated **OR** non-associative organization
1.46	Poorly motivated **AND** non-associative organization

Note that "team" combines the impact of the physical environment with the ability to communicate effectively[48]. A team, according to Webster's dictionary, is a group of people working together. If physical or management communication barriers exist, a team cannot form. A group of people assigned to a common project does not define a team.

A group of software engineers located in an individual cubicle environment cannot work together due to the limited communications allowed; thus, the highest rating possible in the ACAP ratings is a traditional software organization (1.0). Section 8.4 explains in detail the communication issues, how to rate the communications ability of the software development team, and the impact of the development environment on communications and the capability ratings.

An understanding of communication dynamics is essential to the correct evaluation of personnel capability, both analyst (software engineer) and programmer.

8.2.1.2 Programmer capability

The PCAP rates the programming team performance as shown in Table 8-2. The rating criteria are the same as those used in the ACAP rating. The people evaluated in the PCAP rating are the implementers (coders, programmers, etc.) of the software product.

Table 8-2: Programmer capability ratings

PCAP – Programmer Capability	
Value	Description
0.70	Highly motivated **AND** experienced team organization
0.86	Highly motivated **OR** experienced team organization
1.00	**TRADITIONAL** software organization
1.17	Poorly motivated **OR** non-associative organization
1.42	Poorly motivated **AND** non-associative organization

If the same group of people design the software architecture and implement the solution, you would expect the ACAP and PCAP ratings to be identical. However, it is possible that the group is great at design and poor at implementation, or vice versa. In that case, evaluate the ratings accordingly.

8.2.1.3 Application domain experience

Domain experience is the measure of the knowledge and experience the software team has in a specific application area. The measure is one of *effective* team experience and will be described in greater detail in Section 8.2.1.5. In essence, effective experience has much to do with the ability to communicate within the team as a whole.

Lack of experience has multiple effects on the ability to produce a product. Not knowing what to do is one example. Another, recovering from false starts during the definition and implementation phases, also multiplies the lack of productivity. Greater experience allows the development team to leverage knowledge and gain greater efficiencies. The relative productivity swing due to experience alone is approximately 50 percent.

[48] Communication issues are discussed in detail in Section 8.5

8.2.1.4 Learning curve

The learning curve phenomenon is a factor in each of the five experience ratings: application experience (AEXP), development system experience (DEXP), programming language experience (LEXP), practices and methods experience (PEXP) and target system experience (TEXP).

Section 8 is only concerned with the effort impact of the application (domain) experience rating as part of establishing the developer's basic capability including current experience levels and learning curves. The bulk of the experience ratings with the learning curve effects are individually discussed in Section 9.

The effects of the learning curve are observable in three areas:

- An initial penalty determined by the magnitude of technology change.
- A maximum learning rate that is controlled by the complexity of the application, process, or system.
- The mastery cost will equal *a priori*[49]cost if process efficiency remains unchanged.

The *a priori* cost represents the hypothetical cost, or productivity, in the absence of any penalty due to the introduction of a new application area. Over time, the impact of the new discipline will decrease until the area is mastered as illustrated in Figure 8-4. The mastery cost is not always equal to the *a priori cost*. The mastery cost can be lower if the ending capability is more efficient than the *a priori* capability. *The learning curve penalty is always measured at the start of development.*

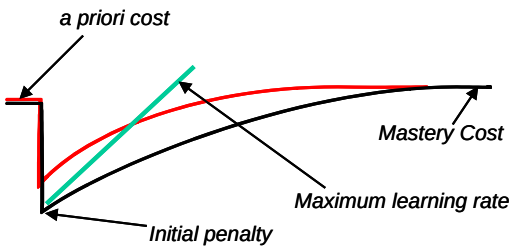


Figure 8-4: Learning curve impact

The five identified learning curve experience parameters are distinguished by two variables, years and complexity. The first variable is the number of years of experience. The second variable is the type or complexity of the language or system. The resulting combination of the two variables donates the relative impact.

The *effective average* experience of the team is an important concept in calculating the cost impact of the experience parameters. Average experience is the simple arithmetic average experience measured over the number of software development team members. If the team members are isolated from each other by placing the members in separate offices or cubicles, they have no opportunity to leverage their learning and experience with the result that the experience rating cannot be better than a simple average. If you place the team members in an environment where the members can leverage their experience, the result is an *effective* experience level that is closer to that of the most experienced member. For example, place an expert in a work area supporting free communications with the requirement that the sole purpose of the expert is to support a group of inexperienced developers. Let the work area be a Skunk Works™. Does the group of developers function as though the group had the experience approaching that of the expert? The relative experience value between a simple average and the expert is almost solely dependent on the quality of communications.

> *No matter what the problem is, it's always a people problem*
>
> G.M. Weinberg, 1988

[49] Proceeding from a known or assumed cause to a necessarily related effect.

8.2.1.5 Domain experience rating

Application experience rates the project impact based upon the effective average application experience for the entire development team. Applications are divided into three categories based upon the task complexity. The categories are: (1) low complexity – application has little or no interaction with the underlying operating system, (2) medium complexity – some interaction with the underlying operating system, as might be found in a typical C2 or C4I system, and (3) high complexity – major interaction with the underlying operating system such as a flight control system.

Development effort penalties—as a function of application complexity and average team domain experience—are shown in Figure 8-5. Note the penalty can be less than one (*a priori* level) indicating a productivity gain for teams with greater experience. This is one example where the *post-mortem* productivity is higher than the *a priori* productivity.

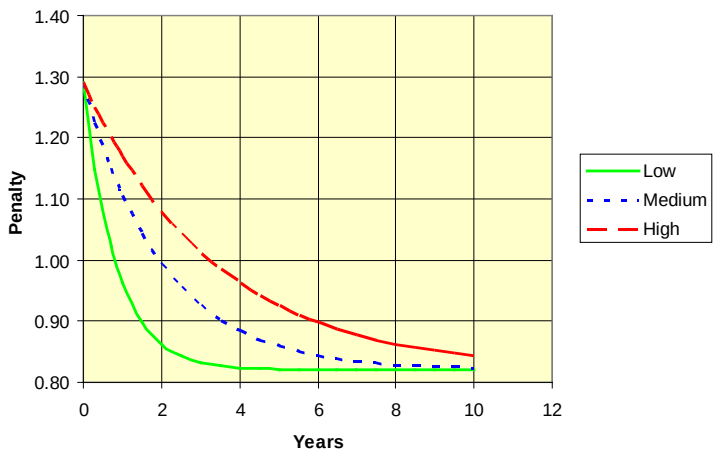


Figure 8-5: Impact of Application Experience on software product development effort

8.2.1.6 Modern practices

The Modern Programming Practices (MODP) rating, shown in Table 8-3, is one of the more difficult environment evaluations. The MODP rating establishes the *cultural* level of modern development methods use in the organization at the *start* of the full-scale software development (Software Requirements Review). Cultural use level defines the level of practice that is the *de facto* development standard; that is, the methods and practices that are used by default in the development. There are other parameters in the environment evaluation that account for less experience with the practices (e.g., development system experience, practices experience). Allowing non-cultural ratings in this parameter evaluation creates double-counting issues. Most project managers claim (at least) reasonable experience in most modern practices. Only demonstrated successful practices use is allowed in the evaluation. Some of the practices considered in this rating are:

Table 8-3: Traditional use of modern practices rating

MODP – Use of Modern Practices	
Value	Description
0.83	Routine use of **ALL** modern practices
0.91	Reasonable experience in **MOST** modern practices
1.00	Reasonable experience in **SOME** modern practices
1.10	**EXPERIMENTAL** use of modern practices
1.21	**NO** use of modern practices

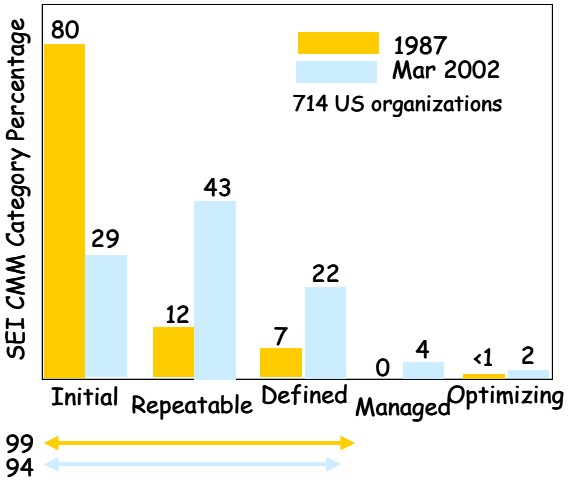
- Object-oriented analysis and design
- Structured design and programming
- Top down analysis and design
- Design and code walkthroughs and inspection
- Pair programming

Note that the list does not include development systems, compilers, etc. These facilities are accounted for in the TOOL rating.

There have been a large number of studies and papers written which cite large productivity gains due to adoption of modern development practices. It

is difficult to isolate the gains to the modern practices from the gains of other factors such as better personnel, better development tools, improved management approaches, better environments, etc.

CMMI levels improve the development process in two significant ways. The first is increased productivity as the organization moves from Level 1 to Level 5. The gain isn't as significant as often advertised, but is better than zero. The second main effect is narrowing the productivity dispersion so the cost and schedule of a development are much simpler to project from historical data. The data shows the trend toward higher CMMI ratings over the 15-year period contained in the report. In the late 1980s, the DoD issued a statement that contracts would not be issued for organizations with less than a Level 3 rating. It is interesting to note that Figure 8-6 shows the trend hits a wall at Level 3 (Defined). Also note in 1987 that 99% of the measured organizations (the yellow line labeled 99) were at Level 3 or below, whereas only 94% were at that level or below in 2002.



Source: Process Maturity Profile of the Software Community 2002 Update – SEMA 3.02

Figure 8-6: CMMI Rating improvement over the period 1987 to 2002

The difficulty in evaluating the MODP rating can be eliminated, or at least reduced, by approaching the measure from another more stable metric, as shown in Table 8-4. The Software Engineering Institute/CMMI rating can be used as a modern practices measure that removes much of the wrangling from the evaluation process. There is strong correlation between the CMMI level and the measurable modern practices rating. Note that once the organization reaches Level 3, further productivity improvement does not materialize until after process data has been collected and used for process optimization/improvement. This improvement may not occur for some years after the Level 5 process maturity has been achieved. Ratings above the "reasonable experience in some modern practices" are not realistic until gains in previous projects can be demonstrated. There are few Level 5 organizations that can justify the improvement needed for the highest two ratings. Note the strong correlations between CMMI and the MODP productivity tables.

Table 8-4: Relationship between CMMI and MODP ratings

MODP – Use of Modern Practices			
CMMI Definition	CMMI Level	Value	Description
Optimizing	5	0.83	Routine use of **ALL** modern practices
Managed	4	0.91	Reasonable experience in **MOST** modern practices
Defined	3	1.00	Reasonable experience in **SOME** modern practices
Repeatable	2	1.10	**EXPERIMENTAL** use of modern practices
Initial	1	1.21	**NO** use of modern practices

8.2.1.7 Modern tools

The automated tool support parameter (TOOL) indicates the degree to which the software development practices have been automated and will be used in the software development. Tools and practices not considered to be part of the development culture are not considered. The list of tools and criteria shown in Table 8-5 can be used to aid in selection of the appropriate value.

Table 8-5: Modern tool categories and selection criteria

Very Low Level of Automation (circa 1950+)	Low Level of Automation (circa 1960+)
Assembler Basic linker Basic batch debugging aids High level language compiler Macro assembler	Overlay linker Batch source editor Basic library aids Basic database aids Advanced batch debugging aids
Nominal Level of Automation (circa 1970+)	**High Level of Automation (circa 1980+)**
Multi-user operating system Interactive source code debugger Database management system Basic database design aids Compound statement compiler Extended overlay linker Interactive text editor Extended program design language Source language debugger Fault reporting system Basic program support library Source code control system Virtual operating system	CASE tools Basic graphical design aids Word processor Implementation standards enforcer Static source code analyzer Program flow and test case analyzer Full program support library with configuration management (CM) aids Full integrated documentation system Automated requirement specification and analysis General purpose system simulators Extended design tools and graphics support Automated verification system Special purpose design support tools
Very High Level of Automation (circa 2000+)	
Integrated application development environment Integrated project support Visual programming tools Automated code structuring Automated metric tools GUI development and testing tools Fourth Generation Languages (4GLs) Code generators Screen generators	

It is common to have tool support with elements in more than one category. In rating the automated tool environment—which likely has tools from more than one category—judgment is necessary to arrive at an effective level of tool support. The judgment should also assess the quality and degree of integration of the tool environment. The rating value should be selected from the Table 8-6 category that best describes the development environment.

Table 8-6: Use of automated tools support rating

TOOL – Automated Tool Support	
Value	Description
0.83	Fully integrated environment (circa 2000+)
0.91	Moderately integrated environment (circa 1980+)
1.00	Extensive tools, little integration, basic maxi tools (circa 1970+)
1.10	Basic mini or micro tools (circa 1960+)
1.24	Very few primitive tools (circa 1950+)

8.2.2 Basic technology constant calculation

The basic technology constant (C_{tb}) describes the *developer's raw capability* unimpeded by the project environment. The basic technology constant was derived by fitting historic project data to the Jensen model (Sage, Seer, and SEER-SEM™) cost and schedule equations. However, the measure is equally valid as a capability measure for the COCOMO family of estimating tools, even though the measure is not automatically produced in these tools (except for REVised Intermediate COCOMO [REVIC]). The resulting basic technology constant equation is:

$$T = ACAP * AEXP * MODP * PCAP * TOOL \qquad (8\text{-}1)$$

$$v = -0.7419 * \ln\left(\frac{T}{4.11}\right)$$

$$C_{tb} = 2000 * \exp(a)$$

The productivity factor for Jensen-based models is the effective technology constant (C_{te}) defined in Equation (5-5). This equation combines the basic technology constant with the product of the development environment factors f_i. The resulting productivity factor is:

$$C_{te} = \frac{C_{tb}}{\prod_i f_i} \qquad (8\text{-}2)$$

8.3 Mechanics of communication

Teams do not function well without effective communication. Two questions to consider in evaluating communication ability are: (1) Does the development area design support free communication between team members? and (2) Are tools in place to support discussion?

Five Commandments for a Highly Productive Environment

I. *Thou shalt not construct communication barriers.*
II. *Thou shalt dedicate the project area.*
III. *Thou shalt not interfere with the project space.*
IV. *Thou shalt provide utensils for creative work.*
V. *Thou shalt not share resources.*

Broadly defined, communication means: the act or process of communicating. It is a process in which information is exchanged between individuals using a common system of symbols, signs, or behavior. The related definition of collaboration is to work jointly with others or together, especially in an intellectual endeavor. Both elements are necessary to effectively produce a software product.

Communication or information transfer is one of the most important considerations in the world of productivity improvement. It dominates a large percentage of the time devoted to software development whether information is transferred via reports, analysis, problem resolution, or training. Several studies suggest that the time spent in some form of communication exceeds 33 percent of a programmer's work day. Improved productivity, therefore, relies on the effective and efficient transfer of information.

The effectiveness of voice or visual radiation is supported by a well-known research study by Mehrabian and Ferris.[50] According to Mehrabian and Ferris, 55 percent of information in presentations is transferred by body language (i.e. posture, gestures, and eye contact), as shown in Figure 8-7. Thirty-eight percent of the information is transferred through vocal tonality (i.e. pitch, volume, etc.), and 7 percent of the information transferred comes from the words, or content, of the presentation. These results are hardly surprising given that our body cues often convey the meaning of our words. For example, we all express many different meanings of the word "no" in normal conversation without giving much thought to the tone and body language accompanying the word.

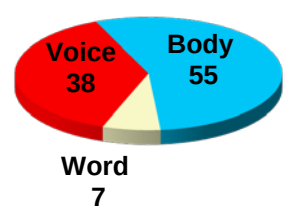


Figure 8-7: Components of communication

The effectiveness of the information transfer, however, is diminished when we remove any source of information radiation. For example, we can remove the visual part of the transfer by forcing the communicators to use a

[50] Mehrabian, A., and S.R. Ferris. "Inference of Attitudes from Nonverbal Communication in Two Channels." *Journal of Counseling Psychology*, Vol. 31, 1967.

telephone. This eliminates all of the gestures, body language, and eye contact from the conversation. These important radiation sources are no longer available to reinforce understanding between the two individuals and can lead to gaps in communication, as well as misunderstandings. For example, we may change our language style when talking on the phone. This could lead to an inference of disinterest, which seeing body language would dispel. People cannot see you nod your head in agreement on the telephone.

The information transfer is further diminished by using e-mail instead of vocal conversation. We eliminate the subtle elements of the conversation radiated by volume and tone such as sarcasm or disappointment. Think of the times you may have called or been called by someone about a date or an appointment and they make an excuse about not being available. The loss of vocal tone may cause you to miss the "get lost" message they are trying to convey.

Information transfer is significantly degraded when we rely solely on paper because we remove the ability to ask and respond to clarifying questions. We lose not only the subtle elements of voice communication, but also the real-time elements necessary for feedback between one another. Feedback may still be present, but at a much slower rate. This impairs the integrity or accuracy of the feedback as well.

> *Companies that sensibly manage their investment in people will prosper in the long run.*
>
> Tom DeMarco and Tim Lister

Some suggest that the solution to communication barriers is modern technological, such as e-mail and network communications. These solutions are often proposed for local communication support and to justify remote software development teams. Ironically, this technological solution raises greater barriers than the cubicle example. At least people in adjacent cubicles have some physical contact. Remote locations are sometimes separated by thousands of miles. The loss of visual and voice radiation, as well as real-time responsiveness, creates a virtual wall.

8.3.1 Information convection

A good analogy for describing communication flow in a development environment was introduced by Alistair Cockburn[51] in 2002. "Convection currents" of information move about a work area just like the movement or dispersion of heat and gas. Air moves freely through an area unless the air is blocked or diverted by an obstruction. Information moves in precisely the same fashion. When two programmers are seated at adjacent desks, they can discuss mutual problems freely, and information flows unobstructed between the two people. The information flow, however, decreases as the programmers' separation distance increases. If a barrier or wall, real or perceived, is placed between the programmers, the information flow is further attenuated, except for the information dispersion that occurs over the wall. If the programmers are placed in private offices, the information flow is blocked and becomes zero. Thus, instead of having the feeling of a team effort, the programmer's attitude becomes "I do my part and then throw it over the wall."

Information flow can also be blocked by a draft, which is a way of describing unwanted or irrelevant information in our metaphor. A draft can be discussion of a topic not related to the project that is loud or obtrusive enough to disturb the other members of a team.

---

[51] Cockburn, A. Agile Software Development. Addison-Wesley, New York, NY: 2002.

8.3.2 Radiation

As previously mentioned, radiation occurs either aurally or visually. Radiation can also occur, on a smaller scale, from touch and smell. Information can also radiate from dry erase boards, paper, posters, sticky notes, and pictures. Because we want to maximize the amount of useful information being conveyed, we will discuss the optimal ways that information is radiated.

The optimal source of radiation communication is both vocal and visual. Voical and visual communication is radiated by expression, gestures, pitch, volume, inflection, exaggerations, and movement. Two people discussing a problem at a dry erase board or at a computer terminal exemplify this ideal situation. This source of radiated information is optimal because of the response time between the speaker's statements and the listener's responses. The real-time nature of the conversation allows instantaneous questions to remove any misunderstandings and to clarify statements and questions.

8.3.3 Communication barriers

As explained, walls impede the flow of information. Consequently, walls decrease productivity. This impediment includes both visible and invisible walls. Assume a large open area filled with workstations that are spaced 10 feet apart front-to-back and side-to-side. People can move freely about the workspace. Since they are not totally enclosed, communication between individuals in this matrix should be reasonably unimpeded[52]. This was the original cubicle concept.

We raise invisible walls if we alternate rows in this matrix with personnel from another project. This spacing causes the distance between related people to increases from 10 to 20 feet. This increased spacing between members of the development team decreases information flow. Thus, the presence of unrelated people forms a literal wall that impedes the information flow. The same effect can be achieved by randomly placing people from a second project in the work area of another project. The information radiated by people from the unrelated second project creates what Cockburn referred to as drafts, a flow of unwanted information.

The optimum information flow communication concept suggests a seating arrangement that increases information flow while discouraging drafts. The project area should be arranged so that people are sitting within hearing distance while limiting information not helpful to them (drafts). You can develop a sense for this as you walk around the development area.

It is difficult to establish analyst and programmer capability ratings higher than "traditional" in the presence of an environment that does not foster effective communications.

8.3.3.1 Skunk Works

A classic example of effective information convection is the Lockheed Skunk Works™, primarily because it dispenses with both physical and non-physical walls. The Skunk Works was an unofficial name given to the Lockheed Advanced Development Projects Unit managed by Kelly Johnson, designer of the SR-71 strategic reconnaissance aircraft. The most successful

[52] Becker, F., and W. Sims. Workplace Strategies for Dynamic Organizations, Offices That Work: Balancing Cost, Flexibility, and Communication. Cornell University International Workplace Studies Program, New York, NY: 2000.

software organizations have followed this paradigm in the organization of their development teams and environments.

As a generic term, “skunk works” dates back to the 1960s. The common skunk works definition is: a small group of experts who move outside an organization’s mainstream operations in order to develop a new technology or application as quickly as possible, without the burden of the organization’s bureaucracy or strict process application. Conventional skunk works operations are characterized by people who are free thinkers, creative, and who don’t let conventional boundaries get in the way (Theory Y). The skunk works workspace is a physically open environment that encourages intra-team access and communication. Tools and processes are tailored and adapted to the project’s requirements. Kelly Johnson established 14 Basic Operating Rules[53] to minimize development risk while maintaining the greatest possible agility and creativity in a lean development team. The rules covered everything from program management to compensation and are relevant for any advanced research unit within a larger organization.

8.3.3.2 Cube farm

A counter-example to the Skunk Works™ approach to software development is the common cube farm. The cube farm violates all of the rules for a productive environment in terms of both communication and collaboration primarily because they raise all the barriers that block effective communication. Unfortunately, the cube farm is the most common, or widely used, software development environment. Probably 90 to 95 percent of the development organizations operating today work in cube farms.

8.3.3.3 Project area

Reiterating the concepts explained above, a physical project area should be allocated for a specific development task and not shared by multiple projects. From the standpoint of information convection, all of the information moving about the development area should be related to the same software development activity. Mixing projects in a specified area creates drafts. Dedicating a specific project area places all of the development personnel in close proximity with as few sources for drafts as possible. Adding people from non-related projects also separates project-related people thereby limiting the information flow and inhibiting discussion and collaboration.

Another side effect of an undedicated project area is that the presence of people from another task prevents the team from forming into a focused, cohesive unit. An extreme view of this phenomenon occurs when the project area is a general software engineering area accommodating multiple projects. Project teams never form in this situation.

8.3.4 Utensils for creative work

We have learned from experience and research that communication and collaboration are key elements in productivity and quality improvement. Our earlier discussion about information convection and radiation suggests a set of low-tech utensils are best for creative work. These utensils include:

[53] Rich, Ben R., and L. Janos. Skunk Works: A Personal Memoir of My Years at Lockheed. Little Brown, Boston: 1994.

- Dry erase boards
- Easel pads
- Butcher paper
- Post-it® notes
- Kitchenette (break room w/dry erase boards)
- Informal discussion areas (brainstorming area)
- Popcorn (the smell attracts people who end up talking, most often about the project they are working on)

None of these utensils fit well within a cubicle environment. Dry erase boards, Post-it® notes, and popcorn can be physically placed in a cubicle, but for *individual* use only. Group activities using the above utensils require large cubicles or workspaces that support teams rather than separate them. The most effective environment supports people working together effectively. Effective team activities require utensils to support communication. You cannot tell children not to eat with their hands without providing an alternative. Likewise, you cannot build project teams without providing team building tools.

When evaluating an organization's productivity, the presence or absence of these tools profoundly affects the result. Tools, experience, environment, management styles, and, especially, communication are important considerations in determining developer capability. They are significant characteristics of the developer organization and, consequently, the basic technology constant which is an important parameter to consider for estimating or when comparing organizations.

The most incomprehensible thing about the world is that it is comprehensible.

Albert Einstein

Section 9

Development Environment Evaluation

Based upon the information described in Section 8, three controlling factor types can be identified which govern software development productivity. These factors are:

1. Basic developer capability
2. *Environment* constraints imposed by the development requirements
3. Product characteristics

The environment factors independent of the development requirements were included in the basic technology constant discussed in Section 8. The environment constraints include the experience levels for programming language, development and target systems, and development practices. Each of the experience factors are subject to the learning curve characteristics introduced in Section 8.2.1.4. The learning curve accounts for the productivity loss due to a lack of expertise by individuals in the environment. People still make mistakes even though efficiency improves with knowledge, understanding, and experience.

> *Einstein argued that there must be a simplified explanation of nature because God is not capricious or arbitrary. No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity.*
>
> Fred Brooks

Product characteristics that limit productivity include the development standards and reliability concerns, requirements volatility, memory and performance constraints, and operational security, among others. Typical productivity (in source lines per person month [ESLOC/PM]) can be as high as 300 ESLOC/PM for an accounting system, yet only 50 ESLOC/PM for a space satellite payload with the variance attributable only to product characteristics.

The environment and development processes contain imperfections. The "volatility" factors describe the impact of these imperfections (bugs). The volatility ratings describe the impact of the environment immaturity. The volatility factors are:

- Development system volatility (DVOL)
- Target system volatility (TVOL)
- Practices/methods volatility (PVOL)

> Poor management can increase software costs more rapidly than any other factor… Despite this cost variation, COCOMO does not include a factor for management quality, but instead provides estimates which assume that the project *will be well managed…*
>
> Barry Boehm

The physical environment also has a DIRECT and significant impact on productivity and quality. The important physical characteristics are:

- Number and proximity of development sites, multiple development sites (MULT)
- Number of real or virtual organizations involved in software development, multiple organizations (MORG)
- Number and severity of security classifications in the project, multiple classifications (MCLS)
- Location of development support personnel, resource support locations (RLOC)
- Access to development and target resources, resource dedication (RDED)

- Average terminal response time (RESP)
- Access to hardcopy – source listings, etc., hardcopy turnaround time (TURN)

9.1 Learning curve vs. volatility

There is an issue that must be clarified before we can discuss the development environment. The impact of the environment on the development cost and schedule occurs on two fronts: learning curve and volatility. The first occurs due to inexperience with the processes and tools present in the environment. The second impact relates to the maturity of the development process itself.

The learning curve, evaluated through the experience parameters, accounts for the loss of productivity due to a lack of understanding and expertise by the individuals in the development environment. Efficiency in any area improves with knowledge, experience, and understanding of that area. People make mistakes; this is a fundamental property of the universe. The number of mistakes, and the impact of those mistakes, can be grouped under an umbrella which we think of as human breakage. The impact of the breakage varies with experience.

Volatility accounts for imperfections in the process. The environment contains "bugs" which we think of as process breakage. As the process matures, the breakage decreases. New technology is *alway*s immature, even if the development organization is certified at CMMI Level 5. Old technology is always *somewhat* immature. Operating systems undergo a continuing series of upgrades during their useful life to repair errors that were undiscovered at the time of delivery. Methods continue to evolve as development methods improve. The evolution of technology never reaches perfection.

Experience and volatility must be accounted for even in mature environments and product lines.

9.2 Personnel experience characteristics

There are three concepts that define the personnel experience characteristics. They are briefly:

1. Initial penalty determined by magnitude of technology change
2. Maximum learning rate controlled by complexity of application, process, or system
3. Mastery cost will equal *a priori* (prior) cost if process efficiency remains unchanged

The learning curve phenomenon is a factor in each of the five experience ratings: application experience (AEXP), development system experience (DEXP), programming language experience (LEXP), practices and methods experience (PEXP), and target system experience (TEXP). Each parameter shows a maximum penalty for no experience at the start of development, and a decreasing penalty based upon prior experience (*always* measured at the start of development) and the relative impact of that parameter. The mastery cost may be less than the *a priori* cost; that is to say that a lot of experience should hopefully improve productivity.

The *a priori cost* represents the hypothetical cost or productivity in the absence of any penalty due to the introduction of the experience parameter of interest. For example, assume you have a "black belt" in the FORTRAN programming language. Use of the language has become an important element of your culture. For this project, you are being introduced to the C++ language due to a project requirement. Both FORTRAN and C++ are third generation languages and, for the purpose of this example, the number of source code lines required to satisfy project requirements are similar for both languages.

The first time you use a language, there will be a productivity loss represented as the initial penalty (shown in Figure 9-1). The second project will have a smaller cost impact. The rate at which you can learn to effectively use the language is the maximum learning rate. The learning rate is a function of the language.

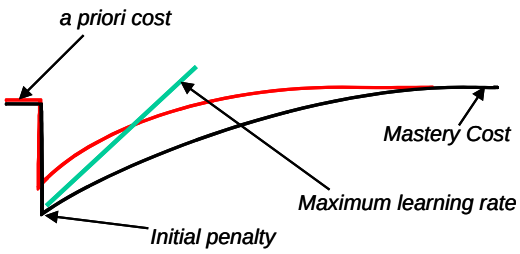


Figure 9-1: Learning curve impact

Over time, the cost impact of the language decreases until the language is mastered. At that time, the use of the language is culture and the impact drops to zero. However, productivity (mastery cost) is near the same as before the language change because the development is unchanged. The mastery cost is not always equal to the *a priori* cost. The mastery cost can be lower if the ending capability is more efficient than the *a priori* capability.

Each of the personnel experience parameters is driven by two variables. The first variable specifies the number of years of experience. The second variable addresses the language or system type or complexity. The result is the relative cost impact defined by the two variables. The impact of programming language experience on software development effort is defined by language complexity and the number of years of experience.

Effective average experience is an important concept in calculating the experience value. Average experience is the simple arithmetic average experience measured over the software development team. If the team members are isolated from each other by placing the members in separate offices or cubicles, they have no opportunity to leverage their learning and experience with the result that the experience rating cannot be better than a simple average. If you place the team members in an environment where the members can work together and leverage their experience, the result is an *effective* experience level that is closer to that of the most experienced member. For example, place an expert in a work area supporting free communications with the requirement that the sole purpose of this expert is to support a group of inexperienced developers. Let the work area be a Skunk Works™. Does the group of developers function as though the group had the experience approaching that of the expert? The relative experience value between a simple average and the expert is almost solely dependent on the quality of communications.

9.2.1 Programming language experience

The LEXP parameter evaluates the effective average experience level for the software development team. The language experience parameter is classified (grouped) in terms of the number of years it takes to master (black belt) the language. Mastery is defined as the level at which a programmer does not need a programming language reference manual to perform most programming tasks. The mastery level has been established from language

studies in the specific programming language. Most programming languages are categorized as one-year languages.

The simplest language to master is the 26 statement Dartmouth BASIC, (see Table 9-1). The next group is the broadest and includes languages that require approximately one year to master. There are two languages requiring three or more years to master. The first, PLI Version F, contains the full language capability including capabilities such as multi-tasking, etc. Note the PL/I language subset G, omitting the special Version F features, requires only one year to master. Ada, C, C++ and C#, using full constracts, have been classed as 5 year mastery languages.

There is a level of programming capability that involves learning to communicate with a computer. Some refer to this ability as Von Neumann[54] thinking. That ability is common to all programming languages. Pascal is a special case when transferring to any complex language. The lower subset of Pascal, Ada and C derivatives are much alike, so that it is assumed that the knowledge gained in the first year of Pascal use is directly transferable to any complex language. Thus, there is a difference between Pascal and FORTRAN as starting languages.

Languages such as Ada, C, C++, C# and others are very complex in their entirety and most developers use a small subset of these powerful languages. If you use the language with its complete structure and capability (including such features as multi-threaded processes, parallel processing and low level bit manipulation), the languages take approximately five years to master. The extra years are not dependent upon the language itself, but are based on the inherent complexity of non-language-specific topics. For example, parallel processing usually requires intimate knowledge of both the language and underlying hardware and operating system characteristics. These complex topics take the same time to master, regardless of the language used.

The relative impact of programming language experience on development cost is shown in Figure 9-2. Each curve in the figure represents one language category from Table 9-1. The worst impact is a relative cost increase of 1.53 for a complex language project using programmers with no experience with the language or only a basic understanding of how to communicate with a computer. The corresponding cost impact for a Dartmouth BASIC project under the same conditions is only 1.15, or 15 percent.

Table 9-1: Programming language mastery time (Years)

Mastery Years	Programming Language
0.5	BASIC
1	FORTRAN COBOL C/ C++ / C# (basic constructs) PL/I Subset G Pascal Ada (basic constructs) Macro Assemblers
2	JOVIAL CMS-2 Mainframe assemblers
3	PL/I Version F
5	Full use of Ada/ C / C++/ C#

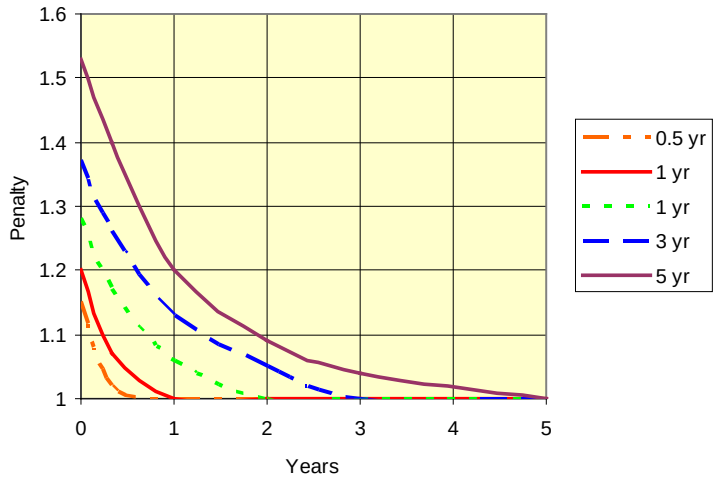


Figure 9-2: Impact of programming language experience on software product development cost

[54] John Von Neumann (1903 – 1957), mathematician and major contributor to a vast range of fields, is considered by many as the father of the modern computer.

9.2.2 Practices and methods experience

The PEXP rating specifies the effective average experience of the team with an organization's current practices and methods at the start of the development task. The groupings (Code 1, etc.) in this parameter are related to the levels of the MODP rating. The rating assumes the organization is moving to a higher MODP rating (1, 2, or 3 level transition). The MODP level the transition is moving "from" is assumed to be a "cultural" MODP level; that is, the organization has been at the "from" level for a long time and it represents the normal organizational approach for developing software. A one-level transition in the MODP rating corresponds to the Code 1 transition.

The organization has zero or more experience at the "to" level. The PEXP experience level specifies the amount of experience gained at the "to" level at the start of software task development. Experience will be gained at this new level during the development. This experience is not considered in evaluating the parameter; in fact, the experience is already accounted for in the parameter values.

If the developer organization is not attempting a transition during the project, the PEXP rating will be 1.0.

An interesting lesson that is visible in Figure 9-3 is the very large penalty involved in attempting a transition from the "Dark Ages" to modern practices makes a large transition dangerous. Any transition from the culture level has a negative productivity impact on the project under consideration and supports the rationale to not introduce any new practices during a development contract.

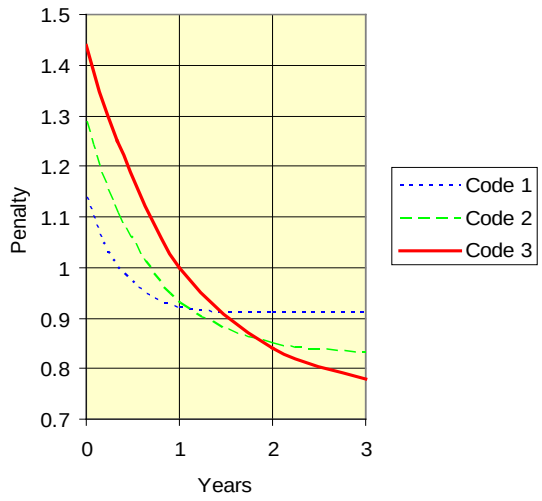


Figure 9-3: Impact of practices and methods experience on software product development effort

9.2.3 Development system experience

The Development System Experience (DEXP) parameter evaluates the impact of the *effective average* development system experience on the project productivity. The development system consists of the hardware and software (computer, operating system, database management system, compilers, and tools) used in the software development. Programming language is NOT considered in this evaluation; however, the compiler and other language tools are considered.

Systems are separated into three groups for the purposes of this evaluation. The first group is the single-user system, such as a desktop personal computer. The second group is a centralized multiple-user system, such as a star configuration that serves the development activity. The third type is a distributed system in which multiple computers and multiple users are involved. Web-type systems are in this category.

Each of the impact curves begins with an experience level of zero years. Experience gained with a single-user system can be applied to the higher complexity systems as demonstrated

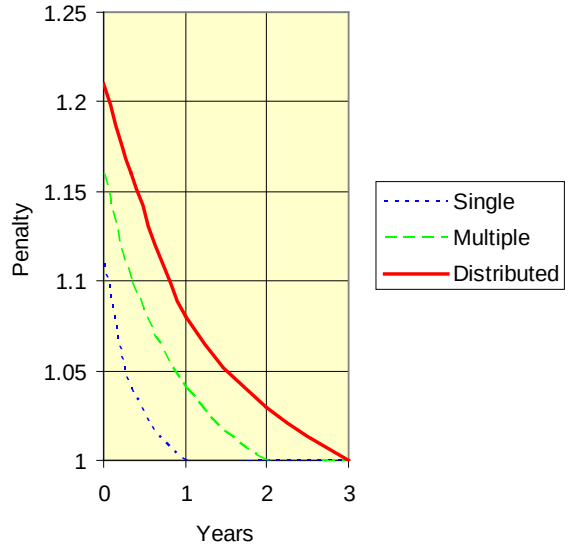


Figure 9-4: Impact of development system experience on software product development effort

for the language experience ratings. As shown in Figure 9-4, a year of experience with a single system reduces the penalty from 1.16, assuming zero experience with the multiple user system, to 1.10 (a one-year penalty). Adding one year of additional multiple user system experience equates to a distributed system experience penalty of 1.03 (two years cumulative experience).

The development system experience rating highlights the high penalty when moving to an unfamiliar development system at the start of a project, even when non-project-related parts of the organization are culturally familiar with the system. It also points out the value of having co-located development system experts as part of the project organization.

9.2.4 Target system experience

The TEXP parameter evaluates the *effective average* target system experience for the project team. The target system, or platform, consists of the hardware and software (computer, operating system, etc.) used in the software product.

Systems are separated into three groups for the purposes of this evaluation (with an example shown in Figure 9-5). The first group is the single-user system, such as a desktop personal computer. The second group is a centralized multiple-user system, such as a star configuration that serves the development activity. The third type is a distributed system in which multiple computers and multiple users are involved. Web-type systems satisfy this category.

Caution: If the target system and development system are the same, set the target system experience level at 3 years (or values to 1.00) so the platform does not get counted twice in the estimate.

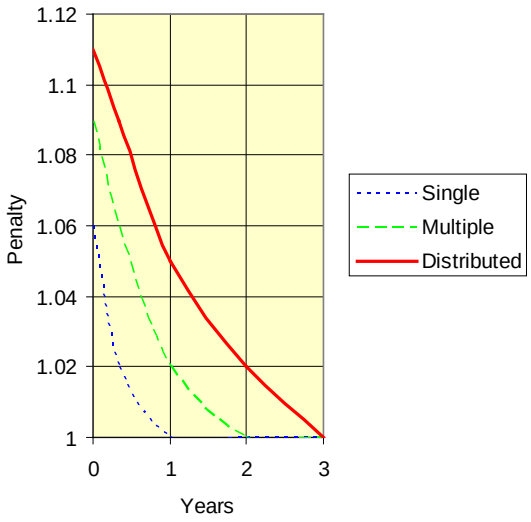


Figure 9-5: Impact of target system experience on software product development effort

9.3 Development support characteristics

9.3.1 Development system volatility

The DVOL parameter accounts for the development effort impact related to problems created or amplified by changes and/or failures in the organization's development system. Those listed as major changes in the parameter are essentially "show stoppers" that halt development until the problems are resolved. Some examples of major changes are: a significant upgrade to an operating system, a re-certified Ada compiler release, or beta versions of any software system.

A minor change creates a slowdown. For example, the slowdown might be caused when an error is uncovered in the process, tools, or methods used in the software development.

The performance penalty accounts for a productivity loss, not a change in software product functionality or a change in product size. The relative impact varies from zero (rating value of 1.00) to a 25 percent increase in development effort, as shown in Table 9-2.

Table 9-2: Development system volatility ratings

DVOL – Development System Volatility	
Value	Description
1.00	**No** major changes, annual minor changes
1.07	**Annual** major changes, monthly minor changes
1.13	**Semiannual** major changes, biweekly minor changes
1.19	**Bimonthly** major changes, weekly minor changes
1.25	**Biweekly** major changes, minor changes every 2 days

9.3.2 Practices/methods volatility

The PVOL rating accounts for problems created or amplified by changes and/or failures in the organization's development practices and methods, as shown in Table 9-3. Those listed as major changes in the parameter are "show stoppers" that halt development until the problems are resolved. For example, the methodology supported by a computer-aided software engineering tool fails to work as advertised, or the vendor releases an upgrade to their tool that changes the way data is tracked.

A minor change creates a slowdown. For example, the slowdown might be caused when a "bug" is uncovered in the process, tools or methods used in the software development.

Table 9-3: Practices/methods volatility ratings

PVOL – Practices/Methods Volatility	
Value	Description
1.00	**No** major changes, annual minor changes
1.08	**Annual** major changes, monthly minor changes
1.15	**Semiannual** major changes, biweekly minor changes
1.23	**Bimonthly** major changes, weekly minor changes
1.30	**Biweekly** major changes, minor changes every 2 days

9.4 Management characteristics

The most important management characteristics (analyst capability, programmer capability, and application domain experience) are included in Section 8 Basic Capability Evaluation.

There are additional environment impacts that are best categorized as management characteristics. These are: (1) multiple levels of security classification in the project environment, (2) multiple development organizations sharing project responsibility, (3) project personnel spread over multiple development sites, and (4) the relative location of the development support personnel. The combined impact of these management issues is significant.

9.4.1 Multiple security classifications

The Multiple Security Classification (MCLS) levels rating accounts for inefficiencies encountered when the development team is working at different access levels; for example, if the project is compartmentalized and part of the team lacks clearances, there is an impact on the organization's efficiency. Obviously, it is better to have all team personnel operating with the same project access. The quantitative development effort impacts of security classification are shown in Table 9-4. These impacts represent the impacts that can be directly related to the classification issue. There are additional impacts related to communication problems that are accounted for in the capability ratings and in the multiple development sites factors.

Table 9-4: Multiple security classifications ratings

MCLS – Multiple Security Classifications	
Value	Description
1.00	**Common** security classification
1.06	**Classified** and unclassified personnel
1.10	**Compartmentalized** and uncleared personnel
1.15	**Multiple** compartment classifications

9.4.2 Multiple development organizations

The Multiple Development Organization (MORG) rating evaluates the impact of MORGs on software development productivity. Multiple organizations always arise when mixing personnel from multiple contractors. Multiple organizations within a single organization are possible, and often appear, due to organization rivalry. Single organization, other organizations, prime and subcontractors (single site), and multiple contractors ratings are shown in Table 9-5.

Table 9-5: Multiple development organizations ratings

MORG – Multiple Development Organizations	
Value	Description
1.00	**Single** development organization
1.09	Developer using personnel from **other** organizations
1.15	**Prime and subcontractor** organization – single site
1.25	**Multiple** contractors – single site

For example, software, system, and test engineering groups within a single organization function as separate contractors. Assigning personnel from those organizations to a product-focused organization where the personnel are controlled and evaluated by the product manager will cause

it to function as a single organization. If not evaluated by the product manager, the rating would deteriorate to the personnel from other organizations' rating.

9.4.3 Multiple development sites

The Multiple Development Sites (MULT) rating accounts for inefficiencies related to separation of team elements. The separation within this parameter relates to physical separation. Table 9-6 shows ratings for single sites, as well as multiple sites in close proximity, separations within 1 hour, and separations less than 2 hours.

Table 9-6**:** Multiple development site ratings

MULT – Multiple Development Sites	
Value	Description
1.00	All developers at **single site** within same development area
1.07	Multiple sites within **close proximity**
1.13	Multiple sites within **1 hour** separation
1.20	Multiple sites with **> 2 hour** separation

A common heuristic states that programmers do not like to walk more than 50 feet to ask a question; this helps to define the term "close proximity." This small separation—such as team members spread around the building's software development area with other projects mixed in the office (cubicle) arrangement—is enough to be the difference between a single site and close proximity rating.

Separation is measured in time units rather than distance. The basic measure is "How long does it take to initiate an action related to a problem?" Politics within or between organizations can add to the time separation.

The next rating value, associated with a one-hour separation, includes development teams separated over multiple floors in a facility or located in adjacent buildings.

The high rating refers to teams being spread across a city, such as the case where there are multiple physical sites.

Caution: Do not be tempted to include working across the Internet, with tools such as e-mail, as sites within close proximity. While necessary and sometimes a major value-added, nothing compares to one-on-one, face-to-face contact to get problems solved. This phenomenon is discussed in greater detail in Section 8.3.

9.4.4 Resources and support location

The Resource and Support Location (RLOC) rating accounts for the inefficiencies incurred due to a separation between the development system support personnel and the development team, as shown in Table 9-7. System support personnel include the following:

- Operating system support
- Development tool and practices support
- System hardware support
- Programming language support
- Target system support

Table 9-7: Resources and support location ratings

RLOC – Resources and Support Location	
Value	Description
1.00	Support personnel **co-located** with development personnel
1.12	Developer and support personnel separation **< 30** minutes
1.23	Developer and support personnel separation **< 4** hours
1.35	Developer and support personnel separation **> 6** hours or working with a foreign target system and language

Access can be limited by either physical, organizational, and/or procedural constraints, in turn creating wasted time and effort. The ideal environment places the support personnel within easy access such as within the magic 50 feet of the development team.

Separation is measured in time units rather than distance as in the MULT rating. The basic measure is: “How long does it take to initiate an action related to a problem?”

The worst case (1.35) support conditions exist when the development team is working with a foreign target system (e.g., Japanese) and operating environment with user documentation written in the foreign language. Developing a product on one continental U.S. coast using a joint contractor and development system on the opposite coast can also create the 6-hour separation from problem occurrence to the time the problem reaches the support personnel.

All life is an experiment.

Ralph Waldo Emerson

Section 10

Product Characteristics Evaluation

Product characteristics make up the third group of parameters that describe a software development project. Section 7 describes the size (magnitude) of the project, Section 8 describes the basic capability of the developer which is the core of the environment, and Section 9 describes the product-related environment. The focus of this section is the software product characteristics.

10.1 Product complexity

Complexity (apparent) is defined as the degree to which a system or component's design or implementation is difficult to understand and verify[55]. In other words, complexity is a function of the internal logic: the number and intricacy of the interfaces and the understandability of the architecture and code. The first thing we must understand about complexity is that it has nothing to do with software size. Size is a separate, independent software product dimension.

There are many measures proposed for software complexity. Some, such as McCabe's Complexity Measure[56], are precise and require the source code before the measure can be completed. Putnam[57] empirically noted that when his software database was plotted as K (total lifecycle effort in person years) versus T^3 (development time in years), the data stratified according to the complexity of the software system. The ratio, currently known as the manpower buildup parameter (MBP), was called D for difficulty in the early days of estimating. The majority of estimating tools refer to D as complexity, mathematically stated as:

$$D = \frac{K}{T^3} \qquad (10\text{-}1)$$

Stratification of the values occurs around the system types as shown in Table 10-1, which is a useful rough guide in determining the appropriate D value for specific software types.

Table 10-1: Stratification of complexity data

D	Description
4	Development primarily using microcode. Signal processing systems with extremely complex interfaces and control logic.
8	New systems with significant interface and interaction requirements with larger system structure. Operating systems and real-time processing with significant logical code
12	Application with significant logical complexity. Some changes in the operating system, but little or no real-time processing.
15	New standalone systems developed on firm operating systems. Minimal interface with underlying operating system or other system parts.
21	Software with low logical complexity using straightforward input/output (I/O) and primarily internal data storage.
28	Extremely simple software containing primarily straight-line code using only internal arrays for data storage.

[55] Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY: 1990.

[56] McCabe, T.J. "A Complexity Measure." IEEE Transactions on Software Engineering New York, NY: 1976.

[57] Putnam, L.H. "A General Empirical Solution to the Macro Software Sizing and Estimating Problem." IEEE Transactions on Software Engineering New York, NY: 1978.

There are some distinguishing features of the Putnam complexity data:

- Complexity is the least significant of the principle estimation parameters except for tasks with high complexity (D<12).
- Complexity is the *inverse* of the complexity number.
- High complexity increases schedule and, because of the small team-size allowed, increases productivity.
- Managing a task as though it was more complex—that is, stretching the schedule or using a smaller team—usually equates to higher productivity.

The characteristics are summarized in Figure 10-1.

Barry Boehm proposed a similar but more detailed complexity definition for the COnstructive COst MOdel (COCOMO). By combining the results of the two approaches, a detailed method for determining software system complexity is shown in Table 10-2. The complexity table is divided into four functional categories, each describing a software focus: control, computation, device-dependent (input/output), and data management. The complexity value selection process consists of the following steps:

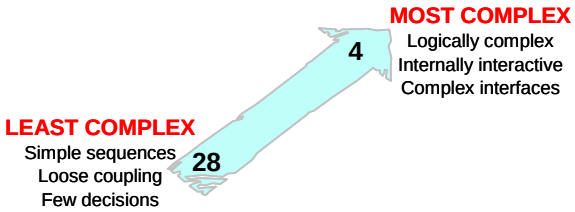


Figure 10-1: Software complexity illustration

1. Select the column that most closely describes the function of the software product. An operating system is largely control. A personnel system is primarily data management.
2. Follow the selected column from a simple system or minimum complexity (28) to most complex or maximum complexity (4), stopping at the row that best describes the function of the system. The operating system is best described by the phrase *Re-entrant and recursive coding. Fixed-priority interrupt handling.* The personnel system is best described by *Multiple input and/or output. Simple structural changes, simple edits.*
3. Select the appropriate complexity value from the Rating column. The operating system description corresponds to a complexity value of 8. The personnel system rating corresponds to a complexity value of 15.
4. If the software system fits multiple categories, follow each appropriate category rating to the row that best describes the function. The rating should correspond to the category that represents the greatest percentage of the software, or in some cases, the category that will drive the development cost and schedule. For example, a weather prediction program is largely one of data presentation; however, the development will likely be driven by the computation aspects of the problem. The best descriptor is "difficult but structured numerical analysis; near singular matrix operations, partial differential equations," which corresponds to a complexity value of eight.

Table 10-2: Complexity rating matrix

Rating	Control Ops	Computational Ops	Device-dependent Ops	Data Mgt Ops
28	Straight line code with a few non-nested Structured Programming (SP) operators; Do's, Cases, If then else's, simple predicates	Evaluation of simple expressions; for example, A=B+C*(D-E)	Simple read/write statements with simple formats	Simple arrays in main memory
21	Straightforward nesting of SP operators. Mostly simple predicates	Evaluation of moderate level expressions; for example, D+SQRT(B**2-4*A*C)	No cognizance needed of particular processor or I/O device characteristics. I/O done at Get/Put level. No cognizance of overlap.	Single-file subsetting with no data structure changes, no edits, no intermediate files.
15	Mostly simple nesting, some inter-module control, decision tables	Use of standard math and statistical routines. Basic matrix and vector operations.	I/O processing includes device selection, status checking and error processing.	Multiple input and/or output. Simple structural changes, simple edits.
12	Highly nested SP operators with many compound predicates. Queue and stack control. Considerable inter-module control.	Basic numerical analysis, multi-variant interpolation, ordinary differential equations. Basic truncation, round-off concerns.	Operations at physical I/O level (physical storage address translations: seeks, reads, etc.) Optimized I/O overlap.	Special purpose subroutines activated by data stream contents. Compiles data restructured at record level.
8	Re-entrant and recursive coding. Fixed-priority interrupt handling.	Difficult but structured Non-linear Analysis (NA): near singular matrix operations, partial differential equations.	Routines for interrupt diagnosis, servicing, masking. Communication line handling.	A generalized, parameter-driven file structuring routine. File building, command processing, search optimization.
4	Multiple-resource scheduling with dynamically changing priorities. Microcode level control.	Difficult and unstructured NA, highly accurate analysis of noisy, stochastic data.	Device timing-dependent coding, micro-programmed operations.	Highly coupled, dynamic relational structures. Natural language data management.

10.2 Display requirements

The Special Display Requirements (DISP) rating is divided into four categories and accounts for the amount of effort to implement user interfaces and other display interactions that cannot be accounted for by size alone. The rating values (penalties) are shown in Table 10-3.

The four DISP categories are:

- *Simple* – no user interface issues
- *User friendly* – Windows or other Graphical User Interface (GUI) software
- *Interactive* – User friendly with the addition of mechanical interfaces such as track balls, etc.
- *Complex* – Interactions with multiple processors such as communication within a flight simulator.

Table 10-3: Special display requirements ratings

DISP – Special Display Requirements	
Value	Description
1.00	**Simple** input/output requirements
1.05	User **friendly**
1.11	**Interactive**
1.16	**Complex** requirements with severe impact

The first category includes software that has no display requirements or only a very simple display interface. The interface is primarily for data presentation with no display interaction.

The second category includes almost all user friendly interfaces. Mouse-driven user interaction is a typical characteristic of these interfaces. The source code necessary to produce the external interface accounts for the bulk of the development effort. The 5 percent penalty accounts for additional design and test effort needed to create and implement the interface. Note that modern programming languages generally develop much of the interface in a building block fashion that eases the implementation and design requirements. Visual Basic and Visual C++ are typical of these languages.

The interactive third category extends the user friendly capability to involve additional mechanical interfaces and sensors to implement the user interaction.

The fourth category accounts for additional effort to implement user interfaces, such as those in flight simulators. This user interface level generally involves multiple processors and system components to support the interface implementation.

10.3 Rehosting requirements

The Rehosting requirements (HOST) rating evaluates the effort to convert the software from the development product to the final product on the target system. This penalty does not apply to projects whose sole purpose is devoted to porting software from one platform, or system, to another. Software porting projects are considered full-scale developments of their own. The values (penalties) are shown in Table 10-4.

Table 10-4: Rehosting requirements ratings

HOST -- Rehosting	
Value	Description
1.00	**No** rehosting – common language and system
1.17	Minor language **OR** minor system change
1.31	Minor language **AND** minor system change
1.63	Major language **OR** major system change
1.94	Major language **AND** major system change

Minor language and/or system changes involve inefficiencies related to minor system differences or minor language differences moving from the development to the target system. A minor change might include moving from FORTRAN IV to FORTRAN 97 or DEC version 5 to version 6. A minor system change could be a move from the contractor's development platform to the target platform in the operational environment if the move is not transparent.

Major changes are "show stoppers," such as moving from JOVIAL to the C++ programming languages or from a Sun platform to an Intel workstation. A major change is not transparent and involves a major software or system development. The 63 percent penalty represents more than an additional 50 percent of the development directly to the target system. If both the language and the system have major development rework, the 94 percent penalty is essentially the cost of developing the software twice.

10.4 Memory constraints

The target system Memory Constraint (MEMC) rating evaluates the development impact of anticipated effort to reduce application memory requirements. The rating is divided into four categories, as shown in Table 10-5.

Table 10-5: Memory constraint ratings

MEMC – Memory Constraint	
Value	Description
1.00	**No** memory economy measures required
1.04	**Some** overlay use or segmentation required
1.15	**Extensive** overlay and/or segmentation required
1.40	**Complex** memory management economy measures required

The first category assumes the available target system memory is abundant and no memory economy measures

are required. Software applications have a way of utilizing all available memory. Not many years ago, the memory limit was only 16 kilobytes. The limit today is in the multi-megabyte range and growing. Many application developments have more than adequate memory resources. No limit means no penalty.

The second category assumes that some single layer overlaying will be required to allow adequate resources with room for growth. The development penalty is only 4 percent.

The third category includes those applications requiring multiple overlay levels or segments. The penalty for the more severe layering is 15 percent.

The final category includes the truly memory-impaired applications in which code and algorithm refinement will be required to satisfy memory constraints. Complex memory management most often occurs in applications where volume and power consumption drive the size of application memory. Spacecraft payloads are typical of the complex memory management applications.

10.5 Required reliability

The Required Reliability (RELY) parameter evaluates the impact of reliability requirements on the ultimate software development effort. The reliability impact includes the specification or documentation level required by the product development, the level of quality assurance (QA) imposed on the development, and the test and rigor required in the unit and formal qualification testing phases. The impacts of reliability requirements are shown in Table 10-6.

Table 10-6: Required reliability ratings

RELY – Required Reliability	
Value	Description
1.00	**Personal** software – user developed
1.16	**Commercial** software – user developed
1.31	MIL-STD with **essential** documentation
1.48	MIL-STD with **full** documentation
1.77	High reliability with **public safety** requirements

The specification level ranges from software for personal use through developments requiring certification or public safety requirements. This list does not include unique very high levels as those required for Minuteman systems, etc. Most software developments fall under the commercial or essential specification levels.

The quality assurance levels do not necessarily equate to the product quality or reliability required. For example, we expect public safety-level QA to be imposed where a loss of life is possible. This includes most Federal Aviation Administration (FAA)-related software, but is not typically required for military aircraft software. Public safety QA is required for nuclear weapons software, but doesn't necessarily apply to nuclear reactor software. The QA level usually matches the corresponding specification level. However, special situations occasionally arise, requiring that the QA level differs from the specification by two or more levels.

The first level (category) assumes the software product will not be used by other than the creator; hence, neither specifications, nor QA, nor testing is required and the impact will be zero.

Level 2 assumes the product will have specifications and that some quality assurance and test (some by the user) will be required. The 16 percent penalty accounts for this product level.

Military Standard (MIL-STD) specifications, quality assurance, and testing are required, according to the software development standards involved in the development. The **IEEE** 12207 standard is typical for this level of reliability. The MIL-STD requirements penalty is about 31 percent. The

penalty can be increased by adding more rigorous QA and test to the development.

Level 4 reflects the penalty associated with the MIL-STD development, with added Independent Verification and Validation (IV&V) requirements. This category includes additional documentation, QA, and test requirements consistent with IV&V.

The highest category includes software developments with public safety requirements similar to those used by the FAA and nuclear systems. The penalty for these developments is near 77 percent. There are requirements that rank above the public safety reliability category. The number of systems above this level is small, too small to define a category, and have very large penalties. The Minuteman missile system is typical of this undefined category.

10.6 Real-time performance requirements

The Real-Time Operation (RTIM) rating evaluates the impact of the fraction of the software product that interacts with the outside environment. The term we use traces back to the real-time system analysis definition. In system analysis, real-time is used to refer to the *control* or *behavior* of the system. Behavior is governed by the sensors, etc., that are external to the system. For example, activate an alarm when the temperature exceeds 451 degrees Fahrenheit. The temperature, the real-time interface, is coupled to an external sensor.

Thus, consider communications with the software to be driven by the external environment or a clock. The fraction of the system that interacts with the external environment will be large in event-driven systems, such as process controllers and interactive systems. Again, the real-time behavior is related to system behavior, not the execution speed of the system. Execution speed is rated by the time constraint parameter (TIMC).

When less than 25 percent of the application software is involved in the real-time behavior, the impact of the real-time operation is negligible as shown in Table 10-7.

When the application software has approximately 50 percent of the software involved in real-time behavior, the effort penalty is 9 percent. When the application real-time involvement reaches approximately 75 percent, the effort penalty is 18 percent. At this level, the application has characteristics of a process controller, or a closely coupled interface between the machine and the environment. The ultimate real-time behavior is reached when the application is totally involved with the external environment. The ultimate real-time penalty is 27 percent, based on available historical data.

Table 10-7: Real time operation ratings

RTIM – Real Time Operation	
Value	Description
1.00	Less than **25%** of source code devoted to real time operations
1.09	Approx **50%** of source code devoted to real time operations
1.18	Approx **75%** of source code devoted to real time operations
1.27	Near **100%** of source code devoted to real time operations

10.7 Requirements volatility

Requirements Volatility (RVOL) is one of the most sensitive cost estimating parameters, both numerically and politically. The RVOL rating evaluates the cost penalty due to expected frequency and scope of requirements changes after the baseline Software Requirements Review, and projects the impact of those changes on delivery cost and schedule. The magnitudes of

the RVOL penalties are shown in Table 10-8. Scope changes do not necessarily result in a change of baseline size. The cost and schedule growth resulting from the RVOL parameter includes the inefficiencies created by the redirections, but does not include impacts due to size changes.

Table 10-8: Requirements volatility ratings

RVOL - Requirements Volatility	
Value	Description
0.93	Essentially **no** requirements changes
1.00	**Familiar** product – small non-critical redirections
1.15	**Known** product – occasional moderate redirections
1.29	Technology **exists** – unfamiliar to developer
1.46	Technology **new** – frequent major redirections

COCOMO uses a similar parameter with the following categories: (1) essentially no changes, (2) small, non-critical redirections, (3) occasional moderate redirections, (4) frequent moderate or occasional redirections, and (5) frequent major redirections. These categories are not as clear as the ones posed in Table 10-8. COCOMO II did away with the cost driver definition, added a new parameter, Requirements Evolution and Volatility (REVL), with a new definition, and moved the parameter to the software product size.

We use a definition that essentially establishes categories that cover the magnitude of the instability with terms that are more appropriate and conceptually easier to grasp. The parameter category definitions are defined in Table 10-8.

The first category specifies that no requirements changes are possible or are improbable. It has occurred more than a few times and is the only product parameter that shows a decrease in development effort. No requirements change is analogous to software being developed on a production line for a well-defined product.

A *familiar* product is a product that has been produced by the developer multiple times and is familiar to the customer providing the requirements. It is an established product, but there will be small requirements changes even though it is an established product line. There is no effort penalty in the product characteristics for this type of development; however, there will be application experience productivity gains in the basic technology area discussed in Section 8.

A *known* product is a product that has been produced by the developer, and the customer provides the requirements at least one time. The developer and the customer understand the technology. The product will likely experience some moderate requirements changes. The requirements changes will increase the size of the software product. The size change is projected by the size growth discussed in Section 7. The 15 percent penalty in the RVOL rating accounts for turbulence in the development process that cannot be accounted for in the size growth.

Technology exists, but is unfamiliar to the developer. It represents the state in which the product has been produced before, but is unfamiliar to the developer and/or the customer providing the product requirements. There will be requirements changes. The effort penalty for this state is 29 percent.

The state in which the *technology is new and unfamiliar* to both the developer and the customer providing the product requirements is true research and development. There will be frequent requirements changes, and some of them will likely be major. The effort penalty for this state is 46 percent, not counting the effort increase that will manifest itself as size growth.

10.8 Security requirements

The Software Security Requirements (SECR) parameter evaluates the development impact of application software security requirements. Apply this parameter (value > 1.00) only if the software product is required to implement the security level specified; that is, if the product is expected not to satisfy, but to implement the specified security level.

The SECR rating is based on the Common Criteria[58], an internationally conceived and accepted security evaluation standard. The effort penalties for each Evaluation Assurance Level (EAL) rating are shown in Table 10-9. The values in the table are for the software development. Testing and certification costs occuring after the completion of development are *not* included in the ratings. High assurance levels above EAL 4 (required for cryptographic certification) will incur additional certification costs.

Table 10-9: Security requirements ratings

SECR – Security Requirements	
Value	Description
1.00	CC EAL 0 – No security requirements (D)
1.03	CC EAL 1 – Functional test
1.08	CC EAL 2 – Structural test (C1)
1.19	CC EAL 3 – Methodical test and check (C2)
1.34	CC EAL 4 – Methodical design, test and review (B1)
1.46	CC EAL 5 – Semiformal design and test (B2)
1.60	CC EAL 6 – Semiformal verification, design and test (B3)
2.35	CC EAL 7 – Formal verification, design and test (A)

The penalties listed in the table apply to situations where the requirements must be satisfied in the software development. It is possible for some of the requirements to be satisfied within an underlying operating system and have no impact on the software itself. For example, software developed for a VAX computing system can use, without explicitly implementing, EAL 3 security assurance.

The holy grail of high assurance for security is EAL 7 because it requires rigorous, formal design and mathematical proof that the security policies of the system are upheld through the development process. Since the EAL 7 rating requires a formal model and proof, a system of more than approximately 5,000 lines of code is very difficult to evaluate. The penalty value for EAL 7 is based on a limited set of historic data and is only approximate. The penalty could be worse than the value given in Table 10-9. On the other hand, the value could be lower, but not likely.

The previous standard used by most estimating systems is based on the National Security Agency Orange Book, which defined security requirements rating on a scale from A through D, with level A at the highest level. The Orange Book ratings are included parenthetically with the (roughly) equivalent CC EAL rating.

[58] The Common Criteria for Information Technology Security Evaluation (abbreviated as Common Criteria or CC) is an international standard (ISO/IEC 15408) for computer security

Appendix A

Acronyms

2GL	Second-Generation Language
3GL	Third-Generation Language
4GL	Fourth-Generation Language
AAF	Adaptation Adjustment Factor
AAP	Abbreviated Acquisition Program
ACAP	Analyst Capability
ACAT	Acquisition Category
ACT	Annual Change Traffic
ACTD	Advanced Concept Technology Demonstration
ADM	Acquisition Decision Memorandum
ADPS	Automatic Data Processing System
AEXP	Application Experience
AFCAA	Air Force Cost Analysis Agency
AFIT	Air Force Institute of Technology
AFMC	Air Force Materiel Command
AFP	Adjusted Function Point
AIS	Automated Information System
AoA	Analysis of Alternatives
APEX	Application Experience (COCOMO II)
API	Application Program Interface
APSE	Ada Programming Support Environment
ASD	Assistant Secretary of Defense
AT&L	Acquisition, Technology, and Logistics
β	Entropy Factor
BF	Backfire Factor (function point conversion)
C4I	Command, Control, Communications, Computers, and Intelligence
C/A	Contract Award
CA	Complexity Attribute
CAD/CAM	Computer-Aided Design/Computer-Aided Manufacturing
CAIG	Cost Analysis Improvement Group
CARD	Cost Analysis Requirements Description
CC	Common Criteria
CD	Concept Design
CDD	Capabilities Description Document
CDR	Critical Design Review
CDRL	Contract Data Requirements List
CER	Cost Estimating Relationships
CIO	Chief Information Officer
CJCSI	Chairman Joint Chiefs of Staff Instruction
CM	Configuration Management
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integration
COCOMO	COnstructive COst MOdel
COTS	Commercial Off-the-Shelf
CPD	Capabilities Production Document
CPLX	Product complexity

CPLX1	PRICE-S parameter combining ACAP, AEXP,CEXP, and TOOL
CPLXM	Management Complexity (PRICE-S)
CPU	Central Processing Unit
CR	Concept Refinement
CSC	Computer Software Components
CSCI	Computer Software Configuration Item
CSU	Computer Software Unit
C_{tb}	Basic technology constant
C_{te}	Effective technology constant
D	Software complexity
DAB	Defense Acquisition Board
DATA	Database size multiplier (COCOMO II)
DAU	Defense Acquisition University
DB	Database
DCAA	Defense Contract Audit Agency
DET	Data Element Type
DEXP	Development System Experience
DISP	Special Display Requirements
DoD	Department of Defense
DoDD	Department of Defense Directive
DoDI	Department of Defense Instruction
DON	Department of the Navy
DRPM	Direct Reporting Program Manager
DRR	Design Readiness Review
D/P	Ratio of bytes in the database to SLOC in the program
DSI	Delivered Source Instructions
DSLOC	Delivered Source Lines of Code
DSYS	Development System Complexity
DVOL	Development System Volatility
EAF	Effort Adjustment Factor
EAL	Evaluation Assurance Level
ECP	Engineering Change Proposal
ECS	Embedded Computer System
EDM	Engineering Development Model
EI	External Inputs
EIF	External Interfaces
EO	External Outputs
EQ	External Inquiries
ERP	Enterprise Resource Program
ESC	Electronic Systems Center (U.S. Air Force)
ESD	Electronic Systems Division
ESLOC	Effective Source Lines of Code
ETR	Effective Technology Rating
EVMS	Earned Value Management System
FAA	Federal Aviation Administration
FAR	Federal Acquisition Regulation
FP	Function Point
FQA	Formal Qualification Audit
FQR	Final Qualification Review
FQT	Formal Qualification Test
FRP	Full-Rate Production
FRPDR	Full-Rate Production Decision Review
FSP	Fulltime Software Personnel

FTR	File Type Reference
FY	Fiscal Year
G&A	General and Administrative
GSC	General System Characteristic
GUI	Graphical User Interface
HACS	Hierarchical Autonomous Communication System
HOL	Higher-Order Language
HOST	Rehosting requirements
HPWT	High Performance Work Teams
HTML	Hypertext Markup Language
HWCI	Hardware Configuration Item
I/O	Input/Output
ICD	Interface Control Document (software development)
	Initial Capabilities Document (acqusition)
ICE	Independent Cost Estimate
IEEE	Institute of Electrical and Electronics Engineers
IFPUG	International Function Point Users Group
ILF	Internal Logical Files
INTEGE	Integration – External (PRICE-S)
INTEGI	Integration – Internal (PRICE-S)
IPT	Integrated Program Team
IOC	Initial Operational Capability
IOT&E	Initial Operation Test and Evaluation
IPD	Integrated Product Development
ISPA	International Society of Parametric Analysts
IV&V	Independent Verification and Validation
JCIDS	Joint Capabilities Integration and Development System
JROC	Joint Requirement Oversight Council
KDSI	Thousands of Delivered Source Instructions
KPP	Key Performance Parameter
KSLOC	Thousand Source Lines of Code
LCCE	Life Cycle Cost Estimate
LEXP	Language Experience
LMSC	Lockheed Missiles and Space Company
LPPM	Lines Per Person-Month
LRIP	Low Rate Initial Production
LRU	Line Replaceable Unit
LTEX	Language and Tools Experience
M	Maturity of the product at the time the estimate is made
MAIS	Major Automated Information System
MBI	Manpower Buildup Index (SLIM®)
MBP	Manpower Buildup Parameter (SLIM®)
MCLS	Multiple Security Classifications
MDA	Milestone Decision Authority
MDAP	Major Defense Acquisition Program
MEMC	Memory Constraint
MIL-STD	Military Standard
MIS	Management Information Systems
MLI	Machine-Level Instruction

MM	Man-Months
MODP	Modern Programming Practices
MORG	Multiple Development Organizations
MULT	Multiple Development Sites
n/a	Not Applicable
NA	Non-linear Analysis
NASA/JSC	National Aeronautics and Space Administration at Johnson Space Center
NEPA	National Environmental Policy Act
NCCA	Naval Center for Cost Analysis
NII	Networks and Information Integration
O&S	Operations and Support
OO	Object-Oriented
OOD	Object-Oriented Design
OS	Operating System
OSD	Office of the Secretary of Defense
OT&E	Operational Test and Evaluation
OTA	Office of Technology Assessment
P&D	Production and Deployment
PCA	Physical Configuration Audit
PCAP	Programmer Capability
PCON	Personnel Continuity
PDL	Program Design Language
PDR	Preliminary Design Review
PEO	Project Executive Officer
PEXP	Practices Experience
PF	Productivity Factor
PI	Productivity Index (SLIM®)
PIMP	Process Improvement parameter
PLAT	Platform parameter (REVIC)
PLTFM	Platform parameter (PRICE-S)
PM	Person-Month
POE	Program Office Estimate
PROFAC	Productivity Factor (PRICE-S)
PS	Processing Step
PSE	Program Support Environment
PVOL	Practices and Methods Volatility Parameter
PWB	Programmer's Workbench
PY	Person-Year
QA	Quality Assurance
QSM	Quantitative Software Management
QUAL	Product quality requirements
RAD	Rapid Application Development
RD	Redesign
RDED	Resource Dedication
RDT&E	Research, Development, Test and Evaluation
RELY	Required Software Reliability
RESP	Terminal response time
RET	Record Element Type
REVIC	Ray's Enhanced Version of Intermediate COCOMO or Revised Intermediate COCOMO
REVL	Requirements Evolution and Volatility
RFP	Request for Proposal

RI	Re-implementation
RLOC	Resource Support Location
ROM	Read-Only Memory
RT	Retest
RTIM	Real-Time Operating Requirements
RUSE	Required Reuse
RVOL	Requirements Volatility
SAF	Size Adjustment Factor
SAR	Software Acceptance Review
SCEA	Society of Cost Estimating and Analysis
SCED	Schedule Constraint
SDD	Software Design Document
SDP	Software Development Plan
SDR	System Design Review
SECNAV	Secretary of the Navy (U.S)
SECR	Software Security Requirements
SECU	DoD security classification parameter (REVIC)
SEER-SEM™	System Evaluation and Estimation of Resources – Software Estimating Model (now known as SEER™ for Software)
SEI	Software Engineering Institute
SER	Size Estimating Relationship
SICAL	Size Calibration Factor
SITE	Multiple development sites parameter (REVIC)
SLIM®	Software Lifecycle Model
SLOC	Source Lines of Code
SMC	Space and Missile Systems Center
SoS	System-of-Systems
SOW	Statement of Work
SP	Structured Programming
SPEC	System Specification Level
SPR	Software Productivity Research
SPS	Software Product Specification
SRR	Software Requirements Review
SRS	Software Requirements Specification
SS	Semantic Statement
SSCAG	Space Systems Cost Analysis Group
SSR	Software Specification Review
STOR	Main storage constraint attribute (COCOMO II)
STP	Software Test Plan
SU	Software Unit
S/W	Software
SWDB	Software Database
SWDM	Software Development Method
SYSCOM	Systems Command
TD	Technology Development
TEMP	Test and Evaluation Master Plan
TEST	Product test requirements
TEXP	Target System Experience
TIMC	Product timing constraints
TIME	Execution time constraint attribute (COCOMO II)
TOC	Total Ownership Cost
TOOL	Automated, modern tool support
TRR	Test Readiness Review
TURN	Hardcopy turnaround time

TVOL	Target System Volatility
UFP	Unadjusted Function Point
UML	Unified Modeling Language
USAF/SMC	U.S. Air Force Space & Missile Systems Center
USD	Under Secretary of Defense
UTIL	Utilization, percentage of hardware memory or processing speed utilized by the Software (PRICE-S), similar to TIMC
V&V	Verification and Validation
VAF	Value Adjustment Factor (function points)
VEXP	Virtual Machine Experience
VHOL	Very High-Order Language
VIRT	Virtual Machine Volatility
WBS	Work Breakdown Structure

Appendix B

Terminology

ADAPTATION ADJUSTMENT FACTOR. Accounts for the effort expended in the development process that does not directly produce product source code. The size adjustment factor is applied to the reused and COTS components of the software system. See also Size Adjustment Factor.

ADJUSTED FUNCTION POINT. Adjusted function points modify the function point count to account for variations in software size related to atypical performance requirements and/or operating environment.

AGILE DEVELOPMENT. The Agile development strategy develops the Computer Software Configuration Item (CSCI) as a continual refinement of the requirements, starting from a simple beginning to a complete product. There are many development approaches that fall under the Agile umbrella.

ALGORITHM. A set of well-defined rules or processes for solving a problem in a definite sequence.

APPLICATION SOFTWARE. Software that implements the operational capabilities of a system.

ARCHITECTURE DESIGN. The second step in the software development process. The architecture contains definitions for each of the Computer Software Components (CSCs) contained in the product. The definition contains the requirements allocated to the CSC and the internal interfaces between the CSCs and the interfaces to the external system. Formal software plans for the CSCs and the higher-level CSCI are also developed in this activity. The activity culminates in a Preliminary Design Review to determine the readiness for advancement to the detail design phase.

ASSEMBLY LANGUAGE. A symbolic representation of the numerical machine code and other constants needed to program a particular processor (CPU).

BACKFIRE. A process used to convert between function points and source lines of code (SLOC). The process produces a functional *total* size.

BASIC TECHNOLOGY CONSTANT. A measure of the raw software development capability of an organization in terms of analyst and programmer capabilities, application experience, use of modern practices and tools, and access to the development system. This constant does not include impacts of the software product-specific constraints. The range of the basic technology constant is between 2,000 and 20,000.

BIT. Unit of information; contraction of binary digit.

BYTE. Abbreviation for binary *term*, a unit of storage capable of holding a single character. A byte can be 5 to 12 bits based on the CPU as an alphabet character. The most common representation of a byte is 8 bits.

CAPABILITY. A major element of the basic technology constant. Capability can be outlined as a measure of motivation, team approaches to development, working environment, problem solving skills, and software engineering ability.

CODE AND UNIT TEST. The fourth step in the software development process. Software Product Specifications are drafted and the production code is written for each Computer Software Unit (CSU). The deliverable source code is thoroughly tested since this is the one level that allows direct access to the code. When the unit testing is successfully completed, the CSU is made available for integration with the parent CSC.

COHESION. An internal property that describes the degree to which elements such as *program statements* and *declarations* are related; the internal “glue” with which a component is constructed. The more cohesive a component, the more related are the internal parts of the component to each other and the overall purpose. In other words, a component is cohesive if all elements of the component are directed toward and essential for performing a single task. A common design goal is to make each component as cohesive as possible or do one thing well and not do anything else.

COMPILER. A program that translates source code into object code. The compiler derives its name from the way it works, looking at the entire piece of source code and collecting and reorganizing the instructions.

COMPLEXITY. The degree to which a system or component has a design or implementation that is difficult to understand and verify. In lay terms, complexity is a function of the internal logic, the number and intricacy of the interfaces, and the understandability of the architecture and code.

COMPONENT-LEVEL ESTIMATE. The component-level (or third-order) estimate is the most comprehensive estimate type. This model combines the effective software product size with a series of environment and product adjustments (defining the development organization production capability and product characteristics) to obtain the development effort or cost. CSCIs are the primary basis of this estimate.

COMPUTER DATA DEFINITIONS. A statement of the characteristics of basic elements of information operated upon by hardware in responding to computer instructions. These characteristics may include – but are not limited to – type, range, structure, and value.

COMPUTER SOFTWARE. Computer instructions or data. Anything that can be stored electronically is software. Software is often divided into two categories: 1) systems software, including the operating system and all the utilities that enable the computer to function, and 2) applications software, which includes programs that do real work for users.

COMPUTER SOFTWARE COMPONENT. A functional or logically distinct part of a CSCI. CSCs may be top-level or lower-level. CSCs may be further decomposed into other CSCs and CSUs.

COMPUTER SOFTWARE CONFIGURATION ITEM. See Configuration Item.

COMPUTER SOFTWARE UNIT. An element specified in design of a CSC that is separately testable.

CONFIGURATION IDENTIFICATION. The current approved or conditionally approved technical documentation for a configuration item as set forth in specifications, drawings, and associated lists; the documents referenced therein.

CONFIGURATION ITEM. Hardware or software, or an aggregation of both, designated by the procuring agency for configuration management. A CSCI is normally defined by a set of requirements documented in a SRS and an ICD or an interface requirements specification.

COTS. Short for ***C****ommercial* ***O****ff-****t****he-****S****helf*, an adjective that describes software or hardware products that are ready-made and available for sale to the general public. COTS software behavior can be characterized in terms of an input set, an output set, and a relationship (hopefully simple) between the two sets. COTS software is a black-box software element for which the countable source code is not known. COTS products are designed to be implemented easily into existing systems without the need for customization.

COUPLING. An external property characterizing the interdependence between two or more modules in a program. Components can be dependent upon each other because of references, passing data, control, or interfaces. The design goal is not necessarily complete independence, but rather to keep the degree of coupling as low as possible.

CSCI ACCEPTANCE TEST. The final step in the full-scale development of the CSCI. The test evaluates the performance of the entire software product (CSCI) against the formal software requirements allocated during the system requirements analysis and approved at the SRR. The CSCI Acceptance Test does not include system integration or user operational tests.

DETAIL DESIGN. The third step in the software development process. The detail design is developed by allocating requirements to each CSC and by establishing the design requirements for each CSU. The activity produces a design for each CSU as well as internal interfaces between CSUs and interfaces to the higher level CSCI. The activity concludes with a formal CDR which reviews the design, test plans, and critical design issues that arise during the activity.

DEVELOPMENT CONFIGURATION. The contractor's software development and associated technical documentation that defines the evolving configuration of a CSCI during the development phase. It is under the development contractor's configuration control and describes the software configuration at any stage of the design, coding, and testing effort. Any item in the Developmental Configuration may be stored on electronic media.

DEVELOPMENT ENVIRONMENT. The environment available to the software development team, including the practices, processes, facilities, development platform, as well as the personnel capability, experience, and motivation.

DEVELOPMENT PLATFORM. Hardware and software tools and computing system.

DRIVER. A piece of software written to accommodate testing of a module before all of its surrounding software is completed. The driver generates all of the inputs for a module. It may do this according to a preprogrammed sequence or be interactive with the tester.

EFFECTIVE SIZE. A measure of the size of the software product that includes the new functionality, the modified functionality, and the energy expended in the development process that does not contribute to the software product. This includes reverse engineering and regression testing of the reused software, and the evaluation and testing of COTS products.

EFFECTIVE SOURCE LINE OF CODE. Incorporates a combination of new SLOC, existing code modified to meet new requirements, and the effort impact of reverse engineering and regression testing of unmodified code and reusable software components.

EFFECTIVE TECHNOLOGY CONSTANT. This constant combines the basic technology constant with the impacts of the development environment affected by the specific software development and constraints on the development contributed by the product characteristics.

EFFICIENCY. A measure of the relative number of machine level instructions produced by a given compiler in processing a given source program.

EMBEDDED COMPUTER SYSTEM (ECS). A computer system that is integral to an electro-mechanical system such as a combat weapon, tactical system, aircraft, ship, missile, spacecraft, certain command and control systems, civilian systems such as rapid transit systems and the like. Embedded computer systems are considered different than automatic data processing systems primarily in the context of how they are developed, acquired, and operated in a using system. The key attributes of an embedded computer system are: (1) It is a computer system that is physically incorporated into a larger system whose primary function is not data processing; (2) It is integral from a design, procurement, or operations viewpoint; and (3) Its output generally includes information, computer control signals, and computer data.

ENTROPY. An index of the degree in which the total energy of a thermodynamic system is uniformly distributed and is, as a result, unavailable for conversion into work. Entropy in software development amounts to the energy expended in the process without contributing to the software product.

EVOLUTIONARY DEVELOPMENT. In evolutionary development, the software requirements for the first version of the product are established; the product is developed and delivered to the customer. At some point during development or after the first product delivery, the requirements for a second, third (and so forth) product release are defined and implemented for the next release.

EXPANSION RATIO. The ratio of a machine level instruction (object code) to higher-order language instructions (source code).

FIRMWARE. Software (programs or data) that has been written into read-only memory (ROM). Firmware is a combination of software and hardware. ROMs, PROMs, and EPROMs that have data or programs recorded on them are firmware.

FOURTH-GENERATION LANGUAGE. Often abbreviated *4GL*, fourth-generation languages are programming languages closer to human languages than typical high-level programming languages. Most 4GLs are used to access databases. The other four generations of computer languages are: First Generation-machine Language, Second Generation-assembly Language, Third Generation-high-level programming Languages such as FORTRAN, C, C++, and Java, and Fifth Generation Languages used for artificial intelligence and neural networks.

FULL-SCALE DEVELOPMENT. All effort expended to development of the software product commencing with the Software Requirements Review (SRR) and continuing through the Final Qualification Test (FQT). This includes architecture development, detail design, code and unit test, internal software integration, and customer acceptance testing. Development time includes the period between the conclusion of SRR through FQT.

FUNCTION POINT. Function Point (FP) analysis is a method for predicting the total SIZE of a software system. FPs measure software size by quantifying the system functionality provided to the estimator based primarily on the system logical design.

GENERAL SYSTEM CHARACTERISTICS. A set of 14 adjustments used to determine the overall Value Adjustment Factor (VAF) in the adjusted function point calculation.

GROWTH. The expected increase in software size during software development. Growth is caused by a number of factors: requirements volatility, projection errors, and functionality changes. Growth is determined by complexity, project maturity, and the distribution of new and reused source code.

HIGHER-ORDER LANGUAGE (HOL). A computer language using human-like language to define machine code operations. A compiler translates HOL statements into machine code (e.g., FORTRAN, COBOL, BASIC, JOVIAL, ADA).

INCREMENTAL DEVELOPMENT. Requirements are allocated functionally to each increment at the beginning of the CSCI development. The full functionality is not available until completion of the final increment. Requirements allocation can be planned to provide partial functionality at the completion of each increment. However, the full CSCI functionality and the ability to test the complete CSCI requirements cannot be completed until all of the incremental developments are complete.

INPUT/OUTPUT (I/O) DRIVER. A piece of software, generally in subroutine format, that provides the unique interface necessary to communicate with a specific peripheral device.

INSTRUCTION. A basic command. The term *instruction* is often used to describe the most rudimentary programming commands. For example, a computer's *instruction set* is the list of all the basic commands in the computer's machine language. An *instruction* also refers to a command in higher-order languages.

INSTRUCTION SET ARCHITECTURE (ISA). The attributes of a digital computer or processor as might be seen by a machine (assembly) language programmer; i.e., the conceptual structure and functional behavior, distinct from the organization of the flow and controls, logic design, and physical implementation.

INTERPRETER. An interpreter translates high-level instructions into an intermediate form, which it then executes. In contrast, a compiler translates high-level instructions directly into machine language. Interpreters are sometimes used during the development of a program, when a programmer wants to add small sections one at a time and test them quickly. In addition, interpreters are often used in education because they allow students to program interactively.

LANGUAGE. A set of symbols, conventions, and rules used to convey information to a computer. Written languages use symbols (that is, characters) to build words. The entire set of words is the language's *vocabulary*. The ways in which the words can be meaningfully combined is defined by the language's syntax and *grammar*. The actual meaning of words and combinations of words is defined by the language's semantics.

LIFE CYCLE. The period of time from the inception of software requirements to development, maintenance, and destruction.

MACHINE LANGUAGE. A first-generation programming language, also called “assembly language;” actual language used by the computer in performing operations; refers to binary or actual codes.

MODIFIED CODE. Pre-developed code that is to be modified before it can be incorporated into the software component to meet a stated requirement.

MODULAR. A software design characteristic that organizes the software into limited aggregates of data and contiguous code that performs identifiable functions. Good modular code has the attributes of low coupling and high internal cohesion.

NEW CODE. Newly developed software.

NON-DEVELOPMENTAL SOFTWARE (NDS). Deliverable software that is not developed under the contract but is provided by the contractor, the government, or a third party. NDS may be referred to as reusable software, government-furnished software, or commercially available software, depending on its source.

OBJECT. Generally, any item that can be individually selected and manipulated. This can include shapes and pictures that appear on a display screen as well as less tangible software entities. In object-oriented programming,

for example, an object is a self-contained entity that consists of both data and procedures to manipulate the data. Objects generally exhibit a behavior defined as functional code.

OBJECT PROGRAM. A first-generation (machine-code) program that can be executed by the computer produced from the automatic translation of a source program.

OPERATING SYSTEM. Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers. For large systems, the operating system is also responsible for security, and ensuring that unauthorized users do not access the system. Operating systems provide a software platform on top of which other programs, called *application programs*, can function. The application programs must be written to run on top of a particular operating system.

OPERATIONAL SOFTWARE. The software that operates in embedded computers.

PERSON-MONTH (PM). The average effort contributed by a single person over a one-month period. A month is assumed to be 152 hours by most estimating tools.

PRODUCTIVITY. The ratio of effective source code size and full-scale development effort. Effort includes all personnel costs attributed to the software project development from the start of architecture design through the completion of the final qualification test. Productivity is sometimes specified in person-hours per ESLOC.

PRODUCTIVITY FACTOR. A multiplication factor (ESLOC/PM) used for projecting software development effort. The factor is a function of product type and effective size. The productivity factor is commonly determined by the product type, historic developer capability, or both, derived from past development projects. The productivity factor is sometimes specified in person-hours per ESLOC.

PRODUCTIVITY INDEX (*PI*). Contains the impacts of product characteristics and the development environment, as well as the domain experience and capability rating for the organization, much like the effective technology constant. *PI* also subsumes the application complexity impact and the effects of software reuse; that is, the impacts of reverse engineering and regression testing are contained in the *PI* value.

PROGRAM. An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner. Without programs, computers are useless. A program is like a recipe. It contains a list of ingredients (called variables) and a list of directions (called statements) that tell the computer what to do with the variables. The variables can represent numeric data, text, or graphical images. Eventually, every program must be translated into a machine language that the computer can understand. This translation is performed by compilers, interpreters, and assemblers.

QUALITY PLAN. Contains an overview of the Quality Assurance and Test Plan, which verifies that the product performs in accordance with the requirements specification and meets all pertinent customer requirements. The Quality Plan is part of the Software Development Plan.

RAYLEIGH STAFFING PROFILE. A roughly bell-shaped curve (shown in graphic) that represents the buildup and decline of development staff in a software lifecycle. An IBM study showed the maximum staffing rate for successful research and development projects never exceeded the maximum staffing rate defined for that type of project. Exceeding the maximum staffing rate correlated strongly with project failure. The decaying exponential curve represents the number of problems remaining to be solved. Combining the two curves produces the Rayleigh staffing profile.

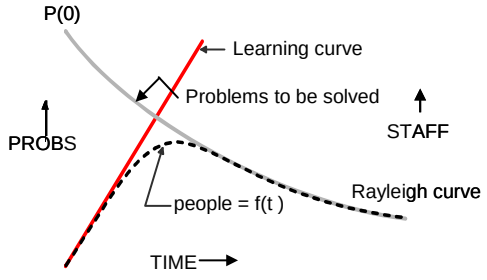


Figure B-1: Rayleigh staffing profile

REAL-TIME. Occurring immediately. Real-time operating systems respond to input immediately. The term evolves from the definition of a real-time system. A real-time system interacts with the outside environment; thus, the term refers to *control* or *behavior* of the system. For example, turn the alarm on when the temperature exceeds 451 degrees Fahrenheit. The sensor is in the external environment, the response is immediate. This definition should not be confused with *time-constrained*, which requires that a process be completed within a specified time.

RELEASE. A configuration management action whereby a particular version of software is made available for a specific purpose (e.g., released for test). Release does not necessarily mean free from defects.

REUSED CODE. Pre-existing code than can be incorporated into a software component with little or no change. The theoretical cost impact of reused code is zero; however, the hidden costs of reverse engineering and regression testing are practical penalties that must be accounted for in reuse.

SECURITY. Software security levels are specified by the Common Criteria Evaluation Assurance Levels (EAL). The assurance levels range from EAL1 for software where security is not viewed as serious to EAL7 for software applications with extremely high risk operation and/or where the value of the assets justifies the high certification costs.

SIZE ADJUSTMENT FACTOR. Accounts for the effort expended in the development process that does not directly produce product source code. The size adjustment factor is applied to the reused and COTS components of the software system. See also Adaptation Adjustment Factor.

SOFTWARE. See Computer Software.

SOFTWARE DESIGN DOCUMENT (SDD). Describes the complete design of a CSCI. It describes the CSCI as composed of Computer Software Components (CSCs) and CSUs. The SDD describes the allocation of requirements from a CSCI to its CSCs and CSUs. Prior to the Preliminary Design Review, the SDD is entered into the Developmental Configuration for the CSCI. Upon completion of the Physical Configuration Audit, the SDD, as part of the Software Product Specification, is entered into the Product Baseline for the CSCI.

SOFTWARE DEVELOPMENT PLAN. Describes the dates, milestones, and deliverables that will drive the development project. It defines who is responsible for doing what and by when. It also describes how the important development activities, such as reviews and testing, will be performed. The activities, deliverables, and reviews are described for each step of the development process.

SOFTWARE DOCUMENTATION. Technical data or information, including computer listings and printouts, which document the requirements, design, or details of the computer software, explains the capabilities and limitations of the software, or provides operating instructions for using or supporting computer software during the software's operational life.

SOFTWARE ENGINEERING ENVIRONMENT. The set of tools, firmware, and hardware necessary to perform the software engineering effort. The tools may include (but are not limited to) compilers, assemblers, linkers, loaders, operating systems, debuggers, simulators, emulators, test tools, documentation tools, and database management systems(s).

SOFTWARE MAINTENANCE. Also known as software support. Maintenance focuses on change associated with error correction, adaptation required as the software's environment evolves, enhancement changes brought about by changing software requirements, and preventative (software re-engineering) changes needed to make the software easier to correct, adapt, or enhance. Each software change degrades the coupling and cohesion of the product making periodic preventative maintenance necessary. Maintenance includes knowledge retention of the software and support environment.

SOFTWARE QUALITY. A judgment by customers or users of a product or service; the extent to which the customers or users believe the product or service meets or surpasses their needs and expectations.

SOURCE LINE OF CODE (SLOC). A measure of software size. A simple, concise definition of a SLOC is any software statement that must be designed, documented, and tested. The three criteria – designed, documented, and tested – must be satisfied in each counted SLOC.

SPIRAL DEVELOPMENT. A risk-driven approach that includes a series of evolving prototypes of the software product culminating in an operational prototype that is the basis for a formal software development.

SUPPORT SOFTWARE. Offline software. For example: development and diagnostic tools, simulation and/or training, maintenance, site support, delivered test software, and report generators.

SYSTEM-LEVEL ESTIMATE. The system level, or first-order, estimate is the most rudimentary estimate type. This model is simply a productivity constant (defining the development organization production capability in terms of arbitrary production units) multiplied by the software product effective size to obtain the development effort or cost.

SYSTEM SOFTWARE. Software designed for a specific computer system or family of computer systems to facilitate the operation and maintenance of the computer system and associated programs. For example: operating system, communications, computer system health and status, security, and fault tolerance.

TRANSACTION. A group of data and operations defined by the application domain. Transactions must have the property of crossing the application boundary. A unique transaction is identified by a unique set of data contents, a unique source and/or destination, or a unique set of operations.

UNADJUSTED FUNCTION POINT (UFP). The UFP count relates only to the total functional size of an average software system with the requirements and structure defined by the FP count. There have been no adjustments for the system type or development and operational environments.

VALUE ADJUSTMENT FACTOR. The mechanism used by the FP methodology to adjust the software size projection in response to special requirements placed on the software.

WATERFALL. The classic software life-cycle model suggests a systematic, sequential approach to software development that begins at the system level and progresses through a series of steps: analysis, design, code, test, integration, and support. The waterfall model outlines steps that are part of the *conventional* engineering approach to solving problems.

WORD. In programming, the natural data size of a computer. The size of a word varies from one computer to another, depending on the CPU. For computers with a 16-bit CPU, a word is 16 bits (2 bytes). On large CPUs, a word can be as long as 64 bits (8 bytes).

Appendix C

Bibliography

Abdel-Hamid, T.K., and S.E. Madnick. "Impact of Schedule Estimation on Software Project Behavior." IEEE Software. July 1986: 70-75.

Abdel-Hamid, T.K., and S.E. Madnick. "Lessons Learned From Modeling the Dynamics of Software Development." Communications of the ACM. 32(12) (1989): 1426-1438.

Abdel-Hamid, T.K. "Investigating the Cost/Schedule Trade-Off in Software Development." IEEE Software. Jan. 1990: 97-105.

Abdel-Hamid, T.K., and S.E. Madnick. Software Project Dynamics: An Integrated Approach. Englewood Cliffs, NJ: Prentice-Hall, 1971.

Acosta, E.O., and D.H. Golub. "Early Experience in Estimating Embedded Software." Journal of Parametrics. 8(1): 69-75.

AFMC Electronic Systems Center. "Cost Analysis: Software Factors and Estimating Relationships." ESCP 173-2B. Hanscom, AFB, MA:1994.

Albanese, Jr., F. The Application of Regression Based and Function Point Software Sizing Techniques to Air Force Programs. Dayton, OH: Air Force Institute of Technology, 1988.

Alberts, D.S, and A.D. Cormier. Software Sizing With Archetypes and Expert Opinion. Proc. of the ISPA Annual Conference. 1987: 458-523.

Albrecht, A.J. Measuring Application Development Productivity. Proc. of the Joint Share/Guide/IBM Application Development Symposium. 1979: 83-92.

Albrecht, A.J., and J.E. Gaffney. "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation." IEEE Transactions on Software Engineering. SE-9 (6) Nov.1983.

Apgar, H. "Software Standard Module Estimating." Journal of Parametrics. 1(4) 1982: 13-23.

Barnes, B.H., and T.B. Bollinger. "Making Reuse Cost-Effective." IEEE Software. Jan. 1991: 13-24.

Basili, V.R., and M.V. Zelkowitz. Analyzing Medium-Scale Software Development. Proc. of the Third International Conf. on Software Engineering. 1978: 116-123.

Basili, V.R., et al. "A Quantitative Analysis of a Software Development in Ada." Univ. Maryland Computer Science Dept. Tech Report. TR-1403 (1984).

Baskette, J. "Life Cycle Analysis of an Ada Project." IEEE Software. 4(1) 1987: 40-47.

Behrens, C.A. "Measuring the Productivity of Computer Systems Development Activities With Function Points." IEEE Trans. Software Eng. Se-9(6) 1983: 648-652.

Belady, L.M., and M.M. Lehman. "Characteristics of Large Systems." Research Directions in Software Technology. Cambridge, MA: MIT Press, 1979.

Bisignani, M.E., and T.S. Reed. "Software Security Costing Issues." Journal of Parametrics. 7(4) 1987: 30-50.

Black, R.K.E., and R. Katz. Effects of Modern Programming Practices on Software Development Costs. Proc. of the IEEE Compcon Conference. Sept. 1977: 250-253.

Boehm, B.W., and P.N. Papaccio. "Understanding and Controlling Software Costs." IEEE Trans. Software Engineering. 14(10). 1988: 1462-1477.

Boehm, B.W., et al. Software Cost Estimation With COCOMO II. Upper Saddle River, NJ: Prentice Hall, 2000.

Boehm, B.W. “A Spiral Model of Software Development and Enhancement.” IEEE Computer. May 1988: 61-72.

Boehm, B.W. “Improving Software Productivity.” IEEE Computer. Sept. 1987: 43-57.

Boehm, B.W. “Industrial Software Metrics Top 10 List.” IEEE Software. Sept. 1987: 84-85.

Boehm, B.W. “Software Engineering.” IEEE Transactions on Computers. Dec. 1976: 1226-1241.

Boehm, B.W. Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall, 1981.

Boehm, B.W. “Software Risk Management: Principles and Practices.” IEEE Software. Jan. 1991: 32-41.

Boehm, B.W. “Software Engineering Economics.” IEEE Transactions on Software Engineering. Se-10(1). 1984: 4-21.

Boehm, B.W. “Understanding and Controlling Software Costs.” Journal of Parametrics. 8(1). 1988: 32-68.

Boehm, B.W., et al. “Cost Models for Future Software Life Cycle Processes: COCOMO 2.0.” Annals of Software Engineering 1. 1995: 295-321.

Booch, Grady. Object-Oriented Analysis and Design With Applications. 2nd ed. Benjamin-Cummings. 1994.

Branyan, E.L, and R. Rambo. “Establishment and Validation of Software Metric Factors.” Proc. ISPA Annual Conference. 1986: 562-592.

Brooks, F.P. The Mythical Man-Month. Reading, MA: Addison-Wesley, 1978.

Cheadle, W.G. “Ada Parametric Sizing, Costing and Scheduling.” Journal of Parametrics. 7(3). 1987: 24-43.

Cheadle, W.G. DoD-STD-2167 Impacts on Software Development. Proc. of the Annual ISPA Conference. 1986: 67-114.

CMMI Product Team. “Capability Maturity Model Integration for Systems Engineering and Software Engineering (CMMI-SE/SW) Staged Representation Version 1.1.” CMU/SEI-2002-TR-002. Software Engineering Institute. Pittsburgh, PA: Dec. 2001.

COCOMO 2.0 Reference Manual (Version 2.0.6). University of Southern California, 1996.

COSTAR User’s Manual (Version 7.00). Amherst, NH: Sofstar Systems. 2008.

Davis, A. Software Requirements Analysis and Specification. Englewood Cliffs, NJ: Prentice-Hall, Inc. 1990.

Defense Acquisition University. “Defense Acquisition Guidebook.” Nov. 2006.

Defense Acquisition University. “Fundamentals of System Acquisition Management (ACQ 101).” Sept. 2005.

DeMarco, Tom. Controlling Software Projects. Englewood Cliffs, NJ: Prentice-Hall, 1982.

DoD Directive 5000.1. “The Defense Acquisition System.” May 12, 2003

DoD Directive 5000.4. “Cost Analysis Group.” Nov. 24, 1992

DoD Instruction 5000.2. “Operation of the Defense Acquisition System.” May 12, 2003.

Dreger, J.B. Function Point Analysis. Englewood Cliffs, N.J: Prentice-Hall, 1989.

Fenick, S. “Implementing Management Metrics: An Army Program.” IEEE Software. 1990: 65-72.

Firesmith, D.G. “Mixing Apples and Oranges or What Is an Ada Line of Code Anyway?” Ada Letters. 8(5). 1988: 110-112.

Fischman, L., et al. Automated Software Project Size Estimation via Use Case Points (Final Technical Report). El Segundo, CA: Galorath, Inc., Apr. 2002.

Fitsos, G.P. “Elements of Software Science.” IBM Tech. Report. TR 02.900, Santa Teresa: IBM, 1982.

Freburger, K., and V.R. Basili. The Software Engineering Laboratory: Relationship Equation. NASA Software Engineering Laboratory. SEL-79-002, 1979.

Funch, P.G. Software Cost Database. Mitre Corporation. MTR 10329, 1987.

Gaffney, Jr., J.E. “An Economics Foundation for Software Reuse.” AIAA Computers in Aerospace Conference VII. 1989: 351-360.

Gaffney, Jr., J.E. “Estimation of Software Code Size Based on Quantitative Aspects of Function (With Application of Expert System Technology).” Journal of Parametrics. 4(3):23. 1984.

Gaffney, Jr., J.E. Software Reuse Economics, Some Parametric Models and Application. Proc. of the ISPA Annual Conf. 1990: 261-263.

Gaffney, Jr., J.E., and Richard Wehrling. Estimating Software Size From Counts of Externals, A Generalization of Function Points. Proc.of the Thirteenth Annual International Society of Parametric Analysts Conference. New Orleans, LA, 1991.

Garmus, David, and David Herron. Function Point Analysis. Boston, MA: Addison-Wesley, 2001.

Garnus, David, ed. IFPUG Function Point Counting Practices Manual (Release 4.2). Westerville, OH: IFPUG, May 2004.

Garvey, P.R. “Measuring Uncertainty in a Software Development Effort Estimate.” Journal of Parametrics. 6(1) 1986: 72-77.

Gayek, J.E., L.G. Long, K.D. Bell, R.M. Hsu, R.K. and Larson. Software Cost and Productivity Model. The Aerospace Corporation (2004).

Gibbs, J. “A Software Support Cost Model for Military Systems.” NATO Symposium. 24-25 Oct. 2001.

Gilb, T. “Estimating Software Attributes. Some Unconventional Points of View.” Journal of Parametrics. 7(1) 1987: 79-89.

Gilb, T. Principles of Software Engineering Management. New York, NY: Addison-Wesley, 1988.

Grady, R.B., and D.L. Caswell. The Use of Software Metrics to Improve Project Estimation. Proc. of the ISPA Annual Conference. (1985): 128-144.

Grady, R.B., and D.L. Caswell. Software Metrics: Establishing A Company-Wide Program. Englewood Cliffs, NJ: Prentice-Hall, 1987: 288.

Grubb, Penny, and A.A. Takang. “Software Maintenance, Concepts, and Practice,” Hackensack, NJ: World Scientific Publishing Co., 2005.

Grucza, Jack. How to Establish a Metrics Database for Software Estimating and Management. Proc. of the ISPA Conference. 1997.

Harmon, B.R., et al. “Military Tactical Aircraft Development Costs, Vol. I Summary Report.” Institute for Defense Analysis, Paper. R-339, 1988.

Helmer, O. Social Technology. New York: Basic Books. 1966.

Herd, J.R., J.N. Postak, We. E. Russell, and K.R. Stewart. “Software Cost Estimation Study – Final Technical Report.” RADC-TR-77-220, Vol. 1, Doty Associates, Inc., Rockville, MD. June 1977.

Hihn, J.M., S. Malhotra, and M. Malhotra. “The Impact of Volatility and the Structure of Organizations on Software Development Costs.” Journal of Parametrics. 10(3), 1990: 65-81.

Holchin, B. Code Growth Study. 23 Dec. 1991.

Humphrey, W.S., W.L. Sweet, et al. “A Method for Assessing the Software Engineering Capability of Contractors.” Software Engineering Institute Tech. Report. CMU/SEI-87-Tr-23, ESD/Tr-87-186, 1987.

Humphrey, Watts S. Managing the Software Process. Reading, MA: Addison-Wesley, 1990.

IEEE-STD-729. IEEE Standard Glossary of Software Engineering Terms. 1983.

IEEE-STD-12207 IEEE Standard Glossary of Software Engineering Terms. 1983.

IEEE. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY: 1990.

International Function Point Users Group. Function Point Counting Practices Manual (Release 4.2). Waterville, OH: International Function Point Users Group, 2004.

ISO/IEC 12207. Standard for Information Technology – Software Life Cycle Processes. 2008.

Itakura, M. and A. Takaynagi. A Model for Estimating Program Size and Its Evaluation. Proc. of the International Conference on Software Engineering. 1982: 104-109.

Jensen, Randall W. A Macrolevel Software Development Cost Estimation Methodology. Proc. of the Fourteenth Asimolar Conference on Circuits, Systems, and Computers. Pacific Grove, CA. Nov. 17-19, 1980.

Jensen, Randall W. Projected Productivity Impact of Near-Term Ada Use in Software System Development. Proc. of the ISPA Annual Conference. 1985: 42-55.

Jensen, Randall W. Sage III Software Schedule and Cost Estimating System User Guide. 2002.

Jensen, Randall W., and Les Dupaix. Commandments for a Productive Software Development Environment. Proc. of the Systems and Software Technology Conference. Salt Lake City, UT, Apr. 18-21, 2005.

Jensen, Randall W. "An Economic Analysis of Software Reuse." CrossTalk. Hill AFB, Ogden STSC, Dec. 2004.

Jensen, Randall W. "Extreme Software Cost Estimating." CrossTalk. Hill AFB, Ogden STSC, Jan. 2004.

Jensen, Randall W. "A Pair Programming Experience." CrossTalk. Hill AFB, Ogden STSC, Mar. 2003.

Jensen, Randall W. "Lessons Learned From Another Failed Software Contract." CrossTalk. Hill AFB, Ogden STSC, Sept. 2003.

Jensen, Randall W. "Software Estimating Model Calibration." CrossTalk. Hill AFB, Ogden STSC, July 2001.

Jensen, Randall W. "Estimating the Cost of Software Reuse." CrossTalk. Hill AFB, Ogden STSC, May 1997.

Jensen, Randall W. "Management Impact on Software Cost and Schedule." CrossTalk. Hill AFB, Ogden STSC, July 1996.

Jensen, Randall W. A New Perspective in Software Cost and Schedule Estimation. Proc. of the Software Technology Conference 1996. Hill AFB, Ogden STSC, 21-26 Apr. 1996.

Jensen, Randall W. Projected Productivity Impact of Near-Term Ada Use in Embedded Software Systems. Proc. of the Seventh Annual International Society of Parametric Analysts Conference. Orlando, FL, May 6-9, 1985.

Jensen, Randall W. A Comparison of the Jensen and COCOMO Schedule and Cost Estimation Models. Proc. of the Sixth Annual International Society of Parametric Analysts Conference. San Francisco, CA, May 17-19, 1984.

Jensen, Randall W. Sensitivity Analysis of the Jensen Software Model. Proce. of the Fifth Annual International Society of Parametric Analysts Conference. St. Louis, MO, Apr. 26-28, 1983.

Jensen, Randall W. An Improved Macrolevel Software Development Resource Estimation Model. Proc. of the Fifth Annual International Society of Parametric Analysts Conference. St. Louis, MO, Apr. 26-28, 1983.

Jones, Capers. Programming Languages Table (Release 8.2). Burlington, MA: Software Productivity Research, Mar. 1996.

Jones, Capers. What Are Function Points. Burlington, MA: Software Productivity Research, Mar. 1995.

Jones, Capers. Applied Software Measurement. New York, NY, McGraw-Hill, 1991.

Jones, Capers. Assessment and Control of Software Risks. Englewood Cliffs, NJ: Prentice-Hall, 1994.

Jones, Capers. Programming Productivity. New York, NY: McGraw-Hill, 1986.

Jones, Capers. "Software Cost Estimation in 2002." CrossTalk. Hill AFB, Ogden STSC, 2002.

Jones, Capers. Estimating Software Costs. Washington, D.C: McGraw-Hill 2007.

Kankey, R.D. "An Overview of Software Maintenance Costing." Estimator. Spring 1991: 40-47.

Kile, R.L. Revic Software Cost Estimating Model User's Manual. Mimeo, Hq AFCMD/EPR, 1994.

Kreisel, G.R. A Software Sizing Approach for Conceptual Programs. Proc. of the ISPA Annual Conf. 1990: 297-310.

Lambert, J.M. "A Software Sizing Model." Journal of Parametrics. 6(4) 1986.

Laranjeira, L.A. "Software Size Estimation of Object-Oriented Systems." IEEE Trans. Software Engineering. 16(5) 1990: 510-522.

Laudon, Kenneth C., and Laudon, Jane P. Management Information Systems: Organization and Technology. 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1997.

Lin, H.-H., and W. Kuo. Reliability Cost In Software Life Cycle Models. Proc. of the Annual Reliability and Maintainability Symposium. 1987: 364-368.

Long, L., K. Bell, J. Gayak, and R. Larson. "Software Cost and Productivity Model." Aerospace Report No. ATR-2004(8311)-1. Aerospace Corporation. El Segundo, CA: 20 Feb. 2004.

Low, G.C., and D.R. Jeffery. "Function Points in the Estimation and Evaluation of the Software Process." IEEE Trans. Software Engineering. 16(1) 1990: 64-71.

Marciniak, John J., and Donald J. Reifer. Software Acquisition Management. New York, NY: John Wiley and Sons, 1990.

McConnell, Steve. Code Complete. 2nd ed. Microsoft Press. Redmond, WA: 2004.

McCabe, Thomas J., and G. Gordon Schulmeyer. System Testing Aided By Structural Analysis. Proc. of the 1982 IEEE COMPSAC. Chicago, IL: Nov. 1982: 523-528.

McGarry, John, et al. Practical Software Measurement. Boston, MA: Addison-Wesley, 2001.

Mills, H.D., M. Dyer, and R. Linger. "Clean Room Software Engineering." IEEE Software Sept. 1987: 19-25.

Mosemann, L.K., II. "Ada and C++: A Business Case Analysis." CrossTalk. Hill AFB, Ogden STSC. Aug./Sept. 1991.

Myers, W. "Allow Plenty of Time for Large-Scale Software." IEEE Software. July 1989: 92-99.

NASA. Goddard Space Flight Center Manager's Handbook for Software Development, Revision 1. NASA Software Engineering Laboratory SEL-84-101, Nov. 1990.

NSIA Management Systems Subcommittee. "Industry Standard Guidelines for Earned Value Management Systems." EVMS Work Team. Aug. 8, 1996.

Norden, P. "Curve Fitting for a Model of Applied Research and Development Scheduling." IBM Journal of Research and Development. Vol. 2 No. 3. July 1958.

Norden, P. "Useful Tools for Project Management." Management of Production. Penguin Books. New Yor, NY: 1970.

Park, R.E. A Price S Profile. Prepared In Support Of IITRI Study For USAF & Army Cost Centers, 1989.

Park, R.E. The Central Equations of the Price Software Cost Model. Mimeo, Moorestown, NJ: G.E. Price Systems 1990.

Park, Robert E. A Manager's Checklist for Validating Software Cost and Schedule Estimates. Carnegie Mellon University, CMU/SEI-95-SR-SR-004, Pittsburgh, PA: Software Engineering Institute, Jan. 1995.

Park, Robert E. "Software Size Measurment: A Framework for Counting Source Statements." <www.sei.cmu.edu/pub/documents/92.reports/pdf/tr20.92.pdf>.

Parkinson, C.N. Parkinson's Law and Other Studies in Administration. Boston, MA: Houghton-Mifflin: 1957.

Perlis, A., F. Sayward, and M. Shaw (Ed.). Software Metrics: An Analysis and Evaluation. Cambridge, MA: MIT Press, 1981.

Pfleeger, Shari Lawrence. "Software Engineering: Theory and Practice," Upper Saddle River, NJ: Prentice-Hall, Inc. 2001.

Pressman, Roger S. Software Engineering: A Practitioner's Approach. 3rd ed. New York, NY: McGraw-Hill, 1992.

PRICE Systems, LLC. PRICE S® Reference Manual. Mt. Laurel, NJ: PRICE Systems, 1998.

PRICE Systems. The Central Equations of the PRICE Software Cost Model. Moorestown, NJ: PRICE Systems, 1988.

Profeta, C.R. Life Cycle Cost Analysis of Evolving Software Systems. Proc. of the ISPA Annual Conference. 1990: 391-409.

Putnam, L.H. "A General Empirical Solution to the Macro Software Sizing and Estimating Problem." IEEE Transactions on Software Engineering. New York, NY: 1978.

Putnam, L.H. A Macro-Estimating Methodology for Software Development. Proc. of the IEEE Compcon 76 Fall. Sept. 1976: 138-143.

Putnam, L.H., and A. Fitzsimmons. "Estimating Software Costs." Datamation. Sept.-Nov. 1979.

Putnam, Lawrence H., and Ware Myers. Fire Core Metrics. New York, NY: Dorset House Publishing, 2003.

Putnam, Lawrence H., and Ware Myers. Measures for Excellence. Englewood Cliffs, NJ: Yourdon Press, 1992.

QSM Corporation. SLIM 3.0 for Windows User's Manual. McLean, VA: Quantitative Software Management, 2003.

REVIC Software Cost Estimating Model User's Manual, Version 9.0. Crystal City, VA: Air Force Cost Analysis Agency, Feb. 1994.

Robertson, L.B., and G.A. Secor. "Effective Management of Software Development." AT&T Technical Journal Mar./Apr. 1986.

Rubin, H.A. "The Art and Science of Software Estimation: Fifth Generation Estimators." Journal of Parametrics. 5(2) 1985: 50-73.

Schultz, H.P. Software Management Metrics. Mitre Corporation (M88-1) for Electronic Systems Division (ESD-Tr-88-001), 1988.

SECNAVINST 5200.2C. "Implementation and Operation of the Defense Acquisition System and the Joint Capabilities Integration and Development System." Nov. 19, 2004

SEER-SEM™ User's Manual (Release 7.0). El Segundo, CA: Galorath, Inc., 2003.

Setzer, R. Spacecraft Software Cost Estimation: Striving For Excellence through Parametric Models (A Review). Proc. of the Aerospace Technology Conference of the Society of Automotive Engineers. TP-851907, 9 1985 (Reprint in Proc. of the ISPA Annual Conference 8:599-607, 1986).

Shalhout, A. Estimating the Cost of Software Maintenance. Proc. of the ISPA Annual Conference. 1990: 324-333.

Shyman, S.R., and T.K. Blankenship. "Improving Methods for Estimating Software Costs." Institute for Defense Analysis (Paper). P-2388, 1990.

Siegel, K.B. "Software Resource Estimating and Program Management." Journal of Parametrics. 10(2), 1990: 62-84.

Software Productivity Research, Inc. CHECKPOINT for Windows User's Guide (Release 2.2.2). Burlington, MA: Software Productivity Research, 1993.

Software Productivity Research. KnowledgePLAN Version 2.0 User's Guide. Burlington. MA: Software Productivity Research, 1997.

Stutzke, Dr. Richard D. "Software Estimating: Challenges and Research." CrossTalk. Hill AFB, Ogden STSC, Apr. 2000: 9-12.

Swanson, E.B. The Dimensions of Maintenance. Proc. of the IEEE/ACM Second International Conf. on Software Engineering. Oct. 1976.

Symons, C.R. “Function Point Analysis: Difficulties and Improvements.” IEEE Trans. Software Engineering. 14(1), 1988: 2-11.

Verner, J., and G. Tate. “Estimating Size and Effort in Fourth-Generation Development.” IEEE Software. July 1988: 15-22.

Wertz, James R., and Wiley J. Larson. Space Mission Analysis and Design. 3rd ed. Kluwer Academic Publishers, 1999.

Wheaton, M.J. “Functional Software Sizing Methodology.” Journal of Parametrics. 6(1) 1986: 17-23.

Whitmire, S. 3D Function Points: Scientific and Real-Time Extensions to Function Points. Proc. of the Tenth Annual Pacific Northwest Software Quality Conference. 1992.

Wolverton, R.W. Airborne Systems Software Acquisition Engineering Guidebook for Software Cost Analysis and Estimating. (ASD-TR-80-5025) Redondo Beach, CA: TRW Corporation, 1980.

Wolverton, R.W. “The Cost of Developing Large-Scale Software.” IEEE Transactions on Computers. June 1974: 615-636.

I'm sorry Dave. I'm afraid I can't do that.
HAL 9000

Appendix D

Software Life Cycle Approaches

There are many representations of a software development life cycle. Each software industry subculture has its own – or several – representations, and each representation tends to be modified somewhat for specific projects. There are many generic approaches that characterize the software development process. Some of the approaches (process models) that are in use today are:

- Waterfall (Section D.1)
- Spiral (Section D.2)
- Evolutionary (Section D.3)
- Incremental (Section D.4)
- Agile (Section D.5)
- Rapid Application Development (Section D.6)
- Other Approaches (Section D.7)

There are certainly additional methods, although the above list covers the mainstream approaches. In the following sections we describe a selected subset of these approaches representing the major underlying models that are pertinent to the purposes of this guidebook.

D.1 Waterfall

The waterfall model is the fundamental basis of most software development approaches and serves well as a basis for describing the typical software development processes. Note the word "typical" allows us to establish steps in a process that are not strictly defined or dictated by Department of Defense (DoD) standards or specifications. This section outlines steps that are part of the *normal* engineering approach to solving problems and are followed in the generalized waterfall development approach common to system development as shown in Figure D-1.

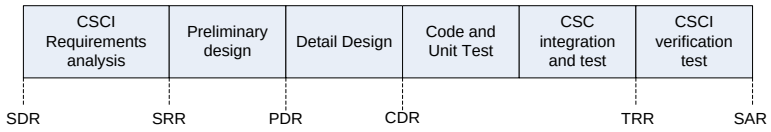


Figure D-1: Software waterfall process

The steps in the *software* waterfall model process are:

- Planning (Software requirements analysis)
- Requirements (CSCI requirements analysis and specification)
- Full-scale development
  - Preliminary design (Architecture design to PDR)
  - Detail design (PDR to CDR)
  - Construction (Code and unit [CSU] development and test)
  - Construction (Component [CSC] integration and test)
  - Construction (Configuration item [CSCI] integration and test)
- System integration and test
- Maintenance

D.2 Spiral development

The spiral development approach[59] shown in Figure D-2 was proposed by B.W. Boehm in 1988. The risk-driven spiral approach includes a series of evolving prototypes of the software product culminating in an operational prototype that is the basis for a formal software development; also known as software development waterfall.

> *In a true spiral development, the cost of developing any software beyond spiral 1 is pure speculation.*
>
> Randall W. Jensen

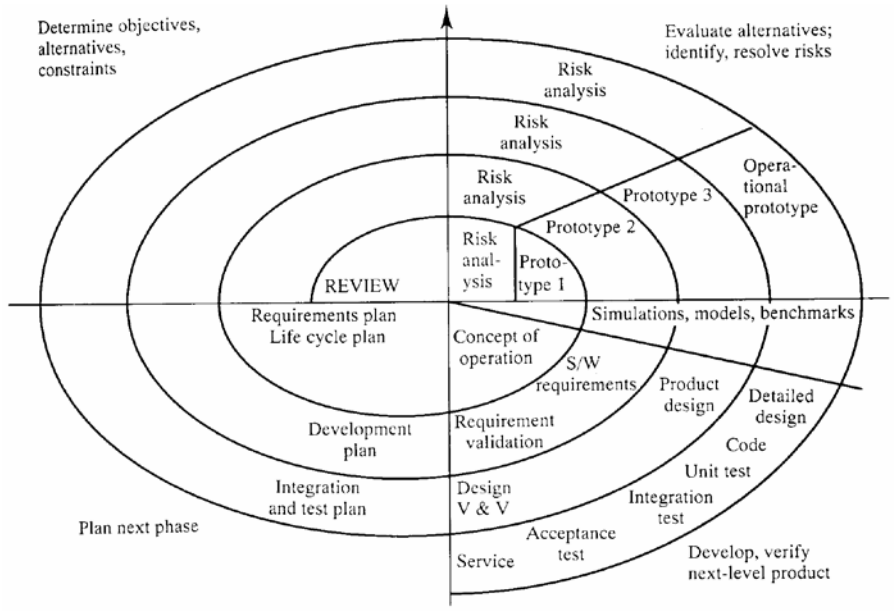


Figure D-2: Spiral development process

The product of the last iteration of the CSCI requirements development is the final operational product prototype. Note the final spiral is similar to a formal traditional engineering waterfall process.

Each iteration of the spiral produces a product. That product can be an undeliverable prototype used to flesh out the definition (capabilities, user interface, etc.) of the final product, or the product can be a deliverable that can be fielded and used as a baseline to be improved in the next spiral of the development.

D.3 Evolutionary development

Evolutionary software development has much in common with the spiral development described in the previous section. In evolutionary development, the software requirements for the first version of the product are established; the product is developed and delivered to the customer. At some point during development or after the first product delivery, the requirements for a second or third etc., product release are defined and implemented for the next release. These evolutions of the product are referred to as evolutionary spirals. Microsoft Word is an example of the development of an evolving product—starting with Word 1, Word 2, and now Word 2007.

> *For both the evolutionary and single-step approaches, software development shall follow an iterative spiral development process in which continually expanding software versions are based on learning from earlier development.*
>
> DoD Instruction 5000.2[4]

[59] Boehm, B.W., "A Spiral Model of Software Development and Enhancement," *Computer*. May, 1988, Pp 61-72

The evolutionary process shown in Figure D-3 is logical and problem-free until the spirals begin to overlap in time. Ideally, there should be no overlap between product developments. When this occurs, the teams implementing the two products must interact to resolve requirements and implementation issues between product requirements for the two spirals and implementations of those spirals. The greater the development overlap, the more significant the impact. A secondary development should not begin until after the preceding development is beyond the detail design milestone (CDR1) because the first product is not stable enough to build upon. The defined functionality of each evolution is only available at its completion.

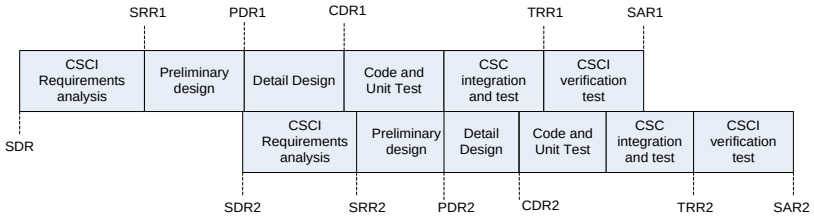


Figure D-3: Evolutionary development process

Each evolution essentially exhibits the characteristics of a waterfall process.

D.4 Incremental development

Incremental developments are characterized by firm requirements allocated at Software Requirements Review (SRR) for each increment (Figure D-4).

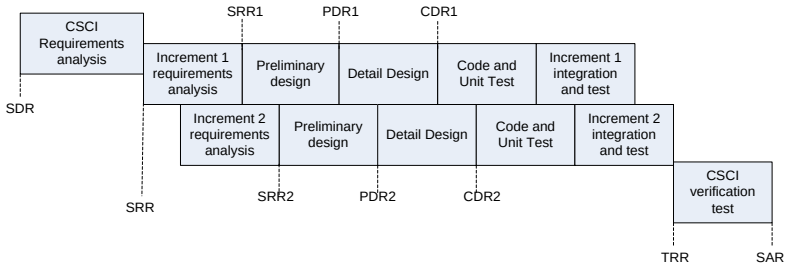


Figure D-4: Incremental software development

Incremental software development should not be confused with either spiral or evolutionary development. The figure shows multiple secondary requirements analysis stages leading to the SRR for each increment. The important point is that the total CSCI requirements are developed and allocated to the individual increments before full-scale development begins.

Full functionality is not available until completion of the final increment as opposed to the evolutionary approach. Requirements allocation can be planned to provide partial functionality at the completion of each increment. However, full CSCI functionality and the ability to test all CSCI requirements cannot be completed until all incremental developments are complete. Partial testing is possible at the completion of each increment.

D.5 Agile development (Extreme programming)

The Agile strategy is to develop the product as a continual refinement of requirements, starting from a simple capability to a complete product. Agile development represents a radical departure from the classic waterfall variation. There are many development approaches that fall under the Agile umbrella. We are going to capture the essence of the concept in this section. This essence is not easy to document, but is extremely easy in concept. The approach starts, as do most of the others, with a set of requirements that have

been approved (at least to a high level). Rather than attempt to develop the full set of requirements in a single large project, the requirements are listed in order of importance and implemented, one at a time, in that order.

The first step in development is to design a test to verify correct implementation of the requirement. The implementation is designed, coded, tested, and reviewed by a small, efficient team. The product builds in size and capability with each iteration. At several points in this repeated process, the product architecture is evaluated and adjusted (re-factored) as necessary to provide an optimum platform for the next iteration. Each iteration adds capability to the product until the product is complete and all requirements are satisfied, as shown in Figure D-5.

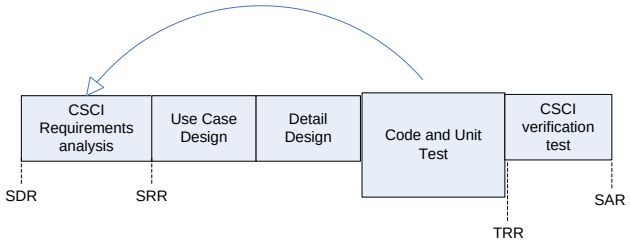


Figure D-5: Agile development

The tests are written from the user's perspective before the code is implemented. They are then collected during the development and run in total at the completion of each iteration. Each refinement is accomplished in days or weeks instead of months and years. At each refinement, the customer can review the progress to ascertain whether the written requirements are consistent with the user's needs and the available technology.

D.6 Rapid application development

Rapid Application Development (RAD) is another approach to the incremental software development process model that emphasizes an extremely short development cycle. The RAD model is an adaptation of the waterfall model in which the rapid, or "high speed," development is achieved through the use of reusable components. The RAD process is primarily used in information system application development. The following process steps are conceptually similar to the traditional waterfall model as follows:

> *Our productivity is great, we just aren't delivering anything.*
>
> Senior Development Manager

1. Build a business function model.
2. Define the data objects that are needed to support the business model.
3. Define the processing necessary achieve the necessary functions using both reusable and new components.
4. Create the software to support the business functions, and integrate it with the reusable components.
5. Test the developed software and system.

D.7 Other approaches

Component-based development incorporates many characteristics of the spiral model. It is evolutionary in nature around the basic spiral. The significant difference between the component and spiral approaches is the use of pre-packaged components, or classes.

The formal methods approach is based on the formal mathematical specification of the computer software. One version of this approach is the cleanroom software engineering method described by Harlan Mills[60].

Fourth-generation techniques encompass a broad spectrum of software tools that allow the developer to specify some characteristics of the software at a higher level. The tools then automatically generate source code to match the user's definition. The form of the process can take on characteristics of the waterfall, the spiral, or the evolutionary models. Care must be taken when using new methods that proper time is spent in verification and validation. The purpose of new methods is not simply to eliminate paperwork or reduce process time – the new method must ensure that the final product meets user needs and maintains high quality.

Harlan D. Mills (1919-1996): Mills' contributions to software engineering have had a profound and enduring effect on education and industrial practices. His cleanroom software development process emphasized top-down design and formal specification. Mills was termed a "super-programmer," a term that would evolve to the concept in IBM of a "chief programmer." In a later era, he might have been called a "hacker."

[60] Mills, H.D., M. Dyer, and R. Linger. "Clean Room Software Engineering." IEEE Software Sept. 1987: 19-25.

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C. Clarke

Appendix E

Software Estimating Models

There are a large number of ways software development estimating models can be categorized. Some estimating models fit into multiple categories, which often lead to difficulties in trying to assign them to a specific taxonomy. Boehm[61] identified seven model categories of which four are useful for comparison. These types are:

1. Analogy
2. Expert judgment
3. Parkinson[62]
4. Price-to-win
5. Top-down
6. Bottom-up
7. Algorithmic (Parametric)

The Parkinson, price-to-win, and top-down types can be eliminated from the list of important methods, not because they don't exist, but because they are undesirable methods. Parkinson's Law states, "Work expands to fill the available volume." Estimates derived from Parkinson's Law tend to be generally accurate because if the estimate leaves a cushion of time and/or money, the product will invariably find "extra" features and testing until the budgeted time and money are consumed. The cushion can also reduce efficiency to consume the budget. The weakness in the Parkinson approach appears when the original budget does not have adequate schedule or money.

The price-to-win (low-bid) method has won many contracts for various contractors. The estimation is based on the customer's budget instead of the software functionality. The method is most often applied for cost plus software and/or system development contracts. The method, applied to a fixed price contract, has driven developers out of business unless the contractor can adjust the fixed price during the contract lifetime. The main reason the price-to-win method is still a common estimating approach is software development estimating technology is immature and software developers are able to convince customers that poor (illegitimate) estimates are valid and achievable.

Law of Definitive Precision

The weaker the data available upon which to base one's conclusion, the greater the precision which should be quoted in order to give that data authenticity.

Norman Augustine, 1983

Top-down software development estimates are derived by allocating software costs from a higher-level product or system budget. The total cost is partitioned from the higher-level budget. Top-down estimates can be derived from any of the seven estimating types listed. In fact, the Parkinson and price-to-win estimates are generally top-down approaches. The major

[61] Boehm, B.W. Software Engineering Economics. Prentice-Hall, Inc., Englewood Cliffs, NJ: 1981, pp. 329-42.
[62] Parkinson, C.N. Parkinson's Law and Other Studies in Administration. Houghton-Mifflin, Boston, MA: 1957.

problem with the top-down estimate is it does not allow the estimator to see the software development requirements in sufficient detail to be effective.

Algorithmic estimating models are based on one or more mathematical algorithms that produce an estimate as a function of a set of variables defined as cost drivers. Parametric models including most of the commercially available estimating tools fall into this category.

The remaining methods are summarized in Table E-1.

Table E-1 Comparison of major software estimating methods

Method	Description	Advantages	Disadvantages
Analogy	Compare project with past projects; scale effort and schedule by historic project data.	Estimate based on actual data and experience.	Few projects are similar enough to use without adjustments.
Expert judgment	Consult or collaborate with one or more "experts."	Little or no historical data is required – useful for new or unique projects.	Experts present their own biases in estimates. Knowledge is often questionable and unsupportable.
Bottom-up	Sum component estimates to product level.	Detailed basis for estimates tend to reduce holes and improve cost projections.	Detailed data is often missing in early development stages. Fails to provide basis for schedule projections.
Parametric	Produce overall project estimate using mathematic algorithms and project characteristics.	CERs based on historical data – allows the estimating model to be adjusted to the proposed development environment and product characteristics. Easy to perform trade-off studies.	The bias from a large number of parameters can lead to inaccurate results without training and experience. Moderately subjective.

E.1 Analogy models

Analogy models are the simplest type of estimating models. They are used to estimate cost by comparing one program with a similar past program or programs, thereby avoiding issues with expert judgment bias. For example, in developing a flight program for a new aircraft, the first estimate step would be to locate an existing flight program from a pre-existing, but similar, aircraft. The pre-existing program has a known size, development cost and schedule, and productivity rate. If the new program has 20 percent more functionality (subjective measure) than the pre-existing program, the cost and schedule for the new program can be projected by scaling the existing cost, ignoring inflation, and scheduling to the new application. The productivity should be approximately constant. The advantage of the analogy method is that it is based on experience. However, the method is limited because, in most instances, similar programs do not exist. For example, you cannot equate the cost of 100,000 lines of Ada code for a bomber's terrain-following program to 100,000 lines of COBOL code for payroll software. Furthermore, for most modern systems, the new programs have no historical precedents.

E.2 Expert judgment models

Expert judgment models or techniques involve consulting with one or more experts to use their knowledge, experience, and understanding of the new project to arrive at a cost and schedule estimate. Experts can factor in differences between historic project data and the requirements of the new software project. The differences often include technology and architecture

changes as well as personnel characteristics, development environment changes, and other project considerations. This method is valuable when no historic precedent exists.

> *Great spirits have always encountered violent opposition from mediocre minds.*
>
> Albert Einstein

On the other hand, expert judgment is often no better than the expertise and objectivity of the estimator, who may be biased (usually optimistic) or unfamiliar with the primary aspects of the development task. It is difficult to find a balance between the quick-response expert estimate (which is often hard to rationalize) and the slower, more thorough estimate provided by an estimating team (group consensus).

Popular expert judgment techniques such as the Delphi and Wideband Delphi methods are often used to support the expert's estimate. Delphi techniques can alleviate bias and experts are usually hard-pressed to accurately estimate the cost of a new software program. Therefore, while expert judgment models are useful in determining inputs to other types of models, they are not frequently used alone in software cost estimation.

E.2.1 Delphi method

Everyone has a bias, including experts. Therefore, it is a good idea to obtain estimates from more than one expert, and there are several ways to combine the estimates into one. The simplest method is to compute the mean of the individual estimates. However, one or two extreme estimates can introduce a bias of their own.

A second method is to lock the experts in a room until they can agree on a single estimate. This method can filter out uninformed estimates but introduces biases of its own. The estimate can be distorted by influential or assertive group members or distorted by authority figures and political issues.

The Delphi Method[63] (or technique, since it is known by both labels) was originated at the RAND Corporation in 1948 as a means of predicting future occurrences, and has become a standard method of forming expert consensus and cost estimation. The Delphi Method can be briefly summarized by the following procedure:

1. Each expert receives a specification (problem statement) and a form to record estimates.
2. Experts fill out forms anonymously. Experts may ask questions of the coordinator, but cannot discuss estimates with each other.
3. The coordinator summarizes the expert's estimates on a form and requests another iteration of the expert's estimate with the rationale behind the new iteration estimate.
4. Experts fill out forms anonymously with the rationale for the estimate.
5. The experts iterate steps 3 and 4 until adequate estimate convergence is achieved. No discussion among experts is allowed until convergence.

[63] Helmer, O. Social Technology. Basic Books, New York, NY: 1966.

E.2.2 Wideband delphi method

Boehm[64] and others concluded the Delphi Method did not provide a wide enough communication bandwidth for the experts to exchange the volume of information necessary to calibrate their estimates with those of the other participants. The Delphi technique was modified to allow this information exchange (Wideband Delphi Method) that is summarized in the following steps:

1. Each expert receives a specification (problem statement) and a form to record estimates.
2. The coordinator chairs a group meeting in which the experts discuss the problem issues with the coordinator and with each other.
3. Experts fill out forms anonymously.
4. The coordinator summarizes the expert's estimates on a form requesting another iteration of the expert's estimate not including the rationale behind estimate.
5. The coordinator calls a group meeting that focuses on the points where the estimates varied.
6. Experts fill out forms anonymously.
7. The coordinator and experts iterate steps 5 and 6 until adequate estimate convergence is achieved. No discussion among experts is allowed until convergence.

E.3 Bottom-up estimating

Bottom-up estimating, also called "grass-roots estimating" or decomposition provides an alternate means of projecting software costs. It involves estimating costs by a detailed analysis of the cost of each unit (or computer software unit [CSU]), then summing unit costs to determine the cost (or effort) for each Computer Software Configuration Item (CSCI), and, possibly, the software cost for the overall system. Bottom-up estimates tend to cover only the costs of developing the individual units; thus, the estimates are often understated. This type of estimate is also more effort intensive than a top-down estimate and is usually more expensive and time consuming.

The most effective way to implement a bottom-up estimate is to establish (organize) a work breakdown structure (WBS) that includes not only the hierarchy of the software product components, but also includes the activity hierarchy of the project elements such as integration, configuration management, and project management. The use of the WBS to collect cost items ensures that all of the cost elements are accounted for.

Bottom-up estimates are often used during proposal preparation and for software project cost-tracking during the development process. This method has the advantage of providing a detailed cost estimate and, consequently, tends to be more accurate and stable than other methods. The method also provides cost tracking, since separate estimates are usually conducted during each software development phase.

Bottom-up estimating has several disadvantages. Since detailed information about each CSU is required, it can be difficult to use during early life cycle

[64] Boehm, B.W. Software Engineering Economics. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981: 335.

phases where detailed information is unavailable. The method is less useful until the detailed product architecture is complete. A second major disadvantage is that the bottom-up method is not intended to produce a schedule estimate—a critical factor in the preparation of a development plan.

E.4 Parametric models

Parametric models, as described here, use one or more mathematical algorithms to produce a software cost and schedule estimate as a function of a number of variables including size, product characteristics, and development environment cost drivers. Parametric models and techniques for software projects generally estimate costs at the system or CSCI levels. The CSCI costs can later be partitioned among lower-level CSUs and/or among life-cycle phases. The advantages of parametric models are that they develop estimates quickly (efficiently), the estimates are repeatable, the estimates encapsulate the estimate basis, and the methods require little detailed information.

Parametric models have weaknesses. The major weaknesses are related to the historic basis of their formulation. A collection of historic data is used to derive the cost estimating relationships (CERs) that are the basis of the model. The models are representations of the data from which they are derived. For example, a model derived from data obtained from schedule-constrained projects is more likely to produce a meaningful estimate for another schedule-constrained project than a model derived from a broad range of projects, including both unconstrained and constrained projects. Parametric models do not generally deal with exceptional condition—such as Agile organizations, exceptional personnel, and management style—unless those conditions have been specifically included in the model's parameters and those parameters have been calibrated to related historic data. Parametric models tend to be less stable than bottom-up methods because they do not inherently foster individual responsibility.

Since most software estimating today is performed using parametric models, it is important to emphasize that no model can compensate for poor size information, poor settings of the model's parameters, lack of estimator experience, or willful misuse.

E.5 Origins and evolution of parametric software models

This section begins with an enlightening trip back into the fuzzy history of computer-based software cost and schedule estimating methods and tools. The first real contribution to the estimating technology occurred in 1958 with the introduction of the Norden staffing profile[65]. This profile, which defines the way staff is assigned to an engineering development, has been directly or indirectly incorporated into most of the estimating methodologies introduced since that time.

The period from 1974 to 1981 brought most of the software estimating models (tools) we use today in the marketplace. The first significant model publication was presented by Ray Wolverton[66] of TRW in 1974. Wolverton

---

[65] Norden, P.V. "Curve Fitting for a Model of Applied Research and Development Scheduling." IBM Journal of Research and Development Vol. 2, No. 3, July, 1958.
[66] Wolverton, R.W. "The Cost of Developing Large-Scale Software." IEEE Transactions on Computers June 1974: 615-636.

was also a major contributor to the development of the COnstructive COst MOdel (COCOMO)[67]. Larry Putnam introduced the Software Lifecycle Model® (SLIM)[68] in 1976, based on his work in the U.S. Army. The third major contribution to the evolution of software estimating tools was the Doty Associates Model[69] developed for the U.S. Air Force in 1977. The Doty Model introduced the concept of cost drivers which was adopted by both Seer and COCOMO. Seer[70] was developed by Dr. Randall W. Jensen at Hughes Aircraft Company in 1979, followed by the publication of COCOMO, a model developed by Barry Boehm and Ray Wolverton at TRW, in 1981. REVIC[71], a recalibration of COCOMO, was introduced in 1988. Seer was commercially implemented as SEER-SEM™ by Galorath Associates in 1990.

Each of these tools evolved slowly (refined algorithms and drivers) until about 1995, when significant changes were made to several models. COCOMO II[72], for example, had several releases between 1995 and 1999. Sage[73], released in 1995, is a major redefinition of the 1979 Seer model that uniquely focuses on management characteristics of the developer as well as the development technology.

When we look at software estimating models from a distance (like 10,000 feet), they generally fall into one of three classes: first-, second- or third-order. The following sections (Sec. E.6 through E.8) describe three common mathematical model classes as a fundamental framework for discussion and comparison.

E.6 First-order models

The first-order model is the most rudimentary parametric estimating model class. This model is, simply, a productivity factor (defining the development organization production capability in terms of arbitrary production units) multiplied by the software product effective size to obtain the development effort or cost. The production units (size) can be source lines of code (SLOC), function points, object points, use cases, and a host of other units depending on one's estimating approach. **The first-order model is the basis for the system-level estimating approach used in this guidebook**.

For the purpose of this discussion, we will use effective source lines of code (ESLOC) as the production unit measure and person hours per effective source line of code as the productivity measure. The first-order estimating model can be summarized as:

---

[67] Boehm, B.W. Software Engineering Economic. Prentice-Hall, Inc., Englewood Cliffs, NJ: 1981.
[68] Putnam, L H. A Macro-Estimating Methodology for Software Development. Proc. IEEE COMPCON 76 Fall. Sept., 1976: 138-143.
[69] Herd, J.R., J.N. Postak, We. E. Russell, and K.R. Stewart. "Software Cost Estimation Study – Final Technical Report." RADC-TR-77-220, Vol. I. Doty Associates, Inc., Rockville, MD: June 1977.
[70] Jensen, R.W. A Macrolevel Software Development Cost Estimation Methodology. Proc. of the Fourteenth Asilomar Conf.on Circuits, Systems, and Computers. Pacific Grove, CA: 17-19 Nov. 1980.
[71] Kile, R.L. REVIC Software Cost Estimating Model User's Manual. HQ AFCMD/EPR, 1988.
[72] B. Boehm, A. Egyed, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby. "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0." Annals of Software Engineering 1995: 295-321.
[73] Jensen, R.W. A New Perspective in Software Cost and Schedule Estimation. Proc. of 1996 Software Technology Conference. Hill AFB, Ogden, UT: 21-26 Apr. 1996.

$$E_d = C_k S_e \qquad \text{(E-1)}$$

where E_d = the development effort in person months (PM),

C_k = a productivity factor (PM/ESLOC), and

S_e = the number of effective source lines of code (ESLOC), or

$$S_e = S_{new} + 0.75 S_{\text{mod}} + 0.2 S_{reused} \qquad \text{(E-2)}$$

where S_{new} = the number of new SLOC being created for the product,
S_{mod} = the number of pre-existing SLOC being modified for the product development, and
S_{reused} = the number of pre-existing, unmodified sloc used in the product.

The productivity factor is commonly determined by the product type, historic developer capability, or both, derived from past development projects. As simple as this equation is, it is widely used to produce high level, rough estimates. This model is essentially the only parametric model that can be effectively used prior to Milestone A because of a lack of available information about the development environment.

By collecting several data points from a specific organization that is developing a specific product type, an average productivity factor can be computed that will be superior to the factors tabulated in this section for specific projects by a contractor. Collecting several data points of different sizes will produce a size-sensitive productivity relationship. Table E-2[14] shows some typical productivity values for various types of software application types.

A system complexity (column D) shows the relationship between the software type and complexity. Complexity is defined and discussed in Section 10.1. It is interesting to note that the productivity rate achieved for each of the software types tends to group around the associated complexity value. Less complex types (higher D values) have higher productivity for each project size group.

Table E-2: Typical productivity factors (person-months per KESLOC) by size and software type

Software Type	D	10 KESLOC	20 KESLOC	50 KESLOC	100 KESLOC	250 KESLOC
Avionics	8	12.6	14.5	17.4	20.0	24.0
Business	15	2.1	2.4	2.9	3.3	4.0
Command and control	10	6.3	7.2	8.7	10.0	12.0
Embedded	8	9.0	10.4	12.4	14.3	17.2
Internet (public)	12	2.1	2.4	2.9	3.3	4.0
Internet (internal)	15	1.1	1.2	1.5	1.7	2.0
Microcode	4-8	12.6	14.5	17.4	20.0	24.0
Process control	12	4.2	4.8	5.8	6.7	8.0
Real-time	8	12.6	14.5	17.4	20.0	24.0

Software Type	D	10 KESLOC	20 KESLOC	50 KESLOC	100 KESLOC	250 KESLOC
Scientific systems/ Engineering research	12	2.5	2.9	3.5	4.0	4.8
Shrink-wrapped/ Packaged	12-15	2.1	2.4	2.9	3.3	4.0
Systems/ Drivers	10	6.3	7.2	8.7	10.0	12.0
Telecommunication	10	6.3	7.2	8.7	10.0	12.0

A simple application of Table E-2 for an effort estimate can be shown as follows:

The effort required for a software development is required for a 50,000 ESLOC avionics system. The effort is given by

$$E_d = C_k S_e$$

= 11.76 * 50 = 588 PM

The corresponding productivity is 50,000/588 = 85 source lines per PM.

E.7 Second-order models

The second-order model compensates for the productivity decrease in larger projects by incorporating an "entropy[74]" factor to account for the productivity change. The entropy effect demonstrates the impact of the large number of communication paths present in large development teams. The development team theoretically has $n(n-1)/2$ communication paths, where n is the number of development personnel. The second-order model becomes

$$E_d = C_{k2} S_e^{\beta} \qquad \text{(E-3)}$$

where β is an entropy factor that measures the communication efficiency in the development team, and

S_e is the number of ESLOC, or

$$S_e = S_{new} + 0.75 S_{\text{mod}} + 0.2 S_{reused} \qquad \text{(E-4)}$$

where S_{new} is the number of new SLOC being created for the product,
S_{mod} is the number of pre-existing SLOC being modified for the product development, and
S_{reused} is the number of pre-existing, unmodified sloc used in the product.

An entropy factor value of 1.0 represents no productivity change with size. An entropy value of less than 1.0 shows a productivity increase with size,

[74] Entropy. An index of the degree in which the total energy of a thermodynamic system is uniformly distributed and is thus unavailable for conversion into work. Entropy in software development amounts to the energy expended in the process without contributing to the software product. For example, reverse engineering of the existing software product is necessary, but produces no product. Communication faults create errors, rework, and consumes resources, but do not contribute to the product.

and a value greater than 1.0 represents a productivity decrease with size. Entropy values of less than 1.0 are inconsistent with historical software data[75]. Most of the widely used models from the 1980s (COCOMO embedded mode, PRICE-S, REVIC, Seer and SLIM) use entropy values of approximately 1.2 for Department of Defense projects. The 1.2 value applies to development tasks using more than 5 development personnel. A lone programmer task has a corresponding entropy factor of 1.0 since there is no communication issue.

Productivity rates are obviously affected by system and CSCI component size as seen in the second- and third-order cost estimating models. The size exponent of approximately 1.2 has a major impact on development effort. The second-order model is included in this appendix for completeness, but is not used in the system- or component-level estimating approaches described in this guidebook.

E.8 Third-order models

The major weakness of the second-order model is its inabilityto adjust the productivity factor to account for variations between projects and differences in development environments. For example, contractor A may have a more efficient process than contractor B; however, contractor A may be using a development team with less experience than assumed in the historic productivity factor. Different constraints may be present in the current development than were present in previous projects. In addition, using a fixed, or calibrated, productivity factor limits the model's application across a wide variety of environments.

The third-order model compensates for the second-order model's narrow applicability by incorporating a set of environment factors to adjust the productivity factor to fit a larger range of problems. The COCOMO form of this model is

$$E_d = C_{k3}\left(\prod_{i=1}^{n} f_i\right) S_e^{\beta} \qquad \text{(E-5)}$$

where f_i = ith environment factor, and

n = number of environment factors.

The number of environment factors varies across estimating models, and is typically between 15 and 32. The factors can be grouped into four distinct types: personnel, management, environment, and product.

A list of the common environment factors used by third-order cost and schedule estimating models is contained in Table E-3. Some of the factors are used by all models with some differences in names and definitions. Analyst capability (ACAP) is an example of factors common to essentially all models. The definition of this parameter varies significantly across these models. COCOMO and Seer (Jensen I) models use a definition based largely on technology. Jensen II models use a definition that emphasizes the impact of management quality as well as technology.

[75] This is a good place to point out a major difference between software and almost any other manufactured product. In other estimating disciplines, a large number of product replications improve productivity through the ability to spread costs over a large sample and reduce learning curve effects. The software product is but a single production item that becomes more complex to manage and develop as the effective size increases.

Schedule constraint (SCED) is a factor used only on COCOMO-based models. This factor is implicit in other models. The Multiple Organizations (MORG) factor is used only in Jensen II-based models.

The third-order model is the basis for the component-level estimating approach used in this guidebook.

Table E-3: Environment factors used by common third-order software cost and schedule estimation models

Characteristic	Acronym	Group	Definition
Analyst Capability	ACAP	Personnel	Measures the impact of front-end development personnel on productivity.
Application Experience	AEXP	Personnel	Measures application experience impact on productivity.
Develop System Experience	DEXP	Personnel	Measures development system experience impact on productivity.
Develop System Volatility	DVOL	Support	Measures development system experience impact on productivity.
Language Experience	LEXP	Personnel	Measures programming language experience impact on productivity.
Memory Constraints	MEMC	Product	Measures the impact of product operating memory requirements on software development cost and schedule.
Modern Practices	MODP	Support	Measures development practices impact on productivity.
Modern Tools	TOOL	Support	Measures impact of software development tools on productivity.
Multiple Classifications	MCLS	Management	Measures impact of security classification separation of development personnel of development productivity.
Multiple Organizations	MORG	Management	Measures the impact of using development personnel, groups, or contactors on development productivity.
Multiple Sites	MULT	Management	Measures the impact of spreading the software CSCI development across multiple locations on productivity.
Practices Experience	PEXP	Personnel	Measures practices and procedures experience impact on productivity.
Process Improvement	PIMP	Support	Measures impact of process improvement change on productivity.
Process Volatility	PVOL	Support	Measures impact of development process and practice volatility on productivity.
Product Complexity	CPLX	Product	Measures the impact of inherent software product complexity on software development cost and schedule.
Product Re-hosting Requirements	HOST	Product	Measures re-hosting impact as part of the development requirements on software development cost and schedule.
Product Reliability Requirements	RELY	Product	Measures the impact of product reliability requirements on software development cost and schedule. This is often divided into quality, documentation and test categories.
Programmer Capability	PCAP	Personnel	Measures the impact of programming and test personnel on productivity.
Real-Time Requirements	RTIM	Product	Measures the impact of the real-time interface requirements with the operational environment on software development cost and schedule.
Required Reuse	RUSE	Support	Measures impact of reuse requirements on development cost and schedule.
Resource Support Location	RLOC	Management	Measures the impact of hardware, system and development tool support personnel isolation on development productivity.
Schedule Constraint	SCED	Support	Measures impact of schedule compression and expansion on development productivity.
Software Security Requirements	SECR	Product	Measures the impact of product security requirements on effort and schedule.

Characteristic	Acronym	Group	Definition
Special Display Requirements	DISP	Product	Measures the impact of user interface requirements on software development and schedule.
Target System Experience	TEXP	Personnel	Measures target system experience impact on productivity.
Target System Volatility	TVOL	Product	Measures the impact of target computer volatility on software development cost and schedule.
Timing Constraints	TIMC	Product	Measures the impact of Central Processing Unit timing constraints on software development cost and schedule.

Tornadoes are caused by trailer parks.
Norm Augustine

Appendix F

System-Level Estimate Case Study

There are two fundamental estimating processes for software development: system and component. The system estimating process is useful early in software acquisition where the software is known at only a high level. At this point in time, the software is defined conceptually and software requirements documents have not been produced. The component-level estimate will be used to perform detailed estimates of software components at the Computer Software Configuration Item (CSCI) level.

The system, being just a concept or only being defined by functional requirements, does not yet have the system components defined. The information available probably includes a very rough size estimate and functional areas (as shown) in Table F-1. From the functional areas, we can derive an approximate productivity value from the software types and the associated system complexity (Cplx) in Tables F-5 and F-6. The estimate developed under these conditions is still only a coarse approximation.

F.1 HACS baseline size estimate

The Hierarchical Autonomous Communication System (HACS) has been created or, rather, translated, as a demonstration vehicle for the cost estimation process. The system contains several functional areas or elements. The baseline size information provided in the Cost Analysis Requirements Document (CARD) specifies that the Control Data Links task has a baseline effective size of 320,000 source lines of code (ESLOC). The CARD also contains enough information to define the User Information Database size in function points. The information at the Software Requirements Review (SRR) is shown in Table F-1.

Table F-1: Baseline description of the HACS Communication System at the software requirements review (concept stage)

Task Name	Size, Eff Baseline	Cplx (*D*)	Units
User Info Database	n/a	13	Function Points
System Manager	219,000	11	ESLOC
Control Data Links	320,000	12	ESLOC
Brain	184,000	10	ESLOC
Sensors	157,000	12	ESLOC
Solitaire	74,000	12	ESLOC
OS (APIs)	53,000	10	ESLOC
System Total	1,012,000	n/a	1,165,149

The CARD places each of the HACS functional areas into the high reliability category with the exception of the Brain (system/driver or process control functions), which falls into the very high category, and Solitaire (no real-time processing), which is at or below low reliability (according to Tables F-5, 6, and 7).

The unadjusted function point (UFP) count for the HACS User Info Database software is calculated in Table F-2. The function point values in the table have been derived from elements in the CARD and the HACS specification. The UFP count for the Info Database task is 84. The process for determining the function point count is explained in Table F-7.

Table F-2: HACS User Info Database unadjusted function point calculation

Component Types	Ranking			Total
	Low	Average	High	
Internal Logical File	1	1	0	17
External Interface File	1	3	0	26
External Input	2	3	0	18
External Output	0	3	0	15
External Inquiry	0	2	0	8
Transform	0	0	0	0
Transition		0		0
Unadjusted Function Points				**84**

The next step in the equivalent baseline software size calculation for the HACS User Info Database is to adjust the function point size to account for differences between the database characteristics and the normal characteristics assumed by the UFP count. This involves adjusting the UFP count by the General System Characteristic (GSC) value adjustment factor (VAF) rating. Once the 14 GSCs for the database have been evaluated, as demonstrated for HACS in Table F-3, the VAF can be computed using the IFPUG VAF equation. The equation is

$$VAF = 0.65 + \left[\left(\sum_{i=1}^{14} GSC_i \right) / 100 \right] \qquad \text{(F-1)}$$

where GSC_i = degree of influence for each GSC, and
i = 1 to 14 representing each GSC.

Table F-3: HACS GSC ratings

No.	General Characteristic	Value	No.	General Characteristic	Value
1	Data communications	0	8	Online update	5
2	Distributed data processing	4	9	Complex processing	0
3	Performance	3	10	Reusability	0
4	Heavily used configuration	4	11	Installation ease	4
5	Transaction rate	5	12	Operational ease	5
6	Online data entry	5	13	Multiple sites	4
7	End-user efficiency	0	14	Facilitate change	4

The resulting VAF score will be between 0.65 and 1.35 depending on the sum of the individual ratings. The sum of the database GSCs is

$$\sum_{i=1}^{14} GSC_i = 43$$

The VAF result for the HACS Info Database is then

$$VAF = 1.08.$$

The adjusted function point (AFP) count is obtained by multiplying the VAF times the UFP count. The standard AFP count is given by:

$$AFP = VAF \times UFP \qquad \text{(F-2)}$$

The AFP count for the HACS Info Database is:

AFP = 1.08 x 84 = 91 function points.

Table F-4: FP to SLOC conversion

Language	SLOC/FP	Range	Language	SLOC/FP	Range
Ada 95	49	40-71	JOVIAL	107	70-165
Assembler, Basic	320	237-575	Object-oriented	29	13-40
Assembler, Macro	213	170-295	Pascal	91	91-160
BASIC, ANSI	32	24-32	PL/I	80	65-95
C	128	60-225	PROLOG	64	35-90
C++	55	29-140	CMS-2	107	70-135
COBOL (ANSI 95)	91	91-175	3rd generation	80	45-125
FORTRAN 95	71	65-150	4th generation	20	10-30
HTML 3.0	15	10-35	Visual Basic 6	32	20-37
JAVA	53	46-80	Visual C++	34	24-40

The conversion of function points (UFP or AFP) to equivalent source lines of code is accomplished by multiplying AFP by a scaling factor (backfire factor [BF]) to produce the total software size as

$$S_t = AFP \times BF \qquad \text{(F-3)}$$

where S_t = total software source lines of code count,

AFP = adjusted function point count, and

BF = SLOC/FP backfire factor from Table F-4.

or,

1.08 x 84 x 55 = 5,000 (approximately) total SLOC

The resulting User Info Database effective size for the HACS software is presented in Table F-5 assuming the implementation language for the User Info Database is C++ and *BF = 55*. The effective size is equal to the total size since there is no pre-existing code assumed in this value.

The baseline effective size is the starting size for the effort analysis. The baseline values from the CARD are summarized in Table F-7 Column 2. This value assumes no software growth during development. At the concept stage, the size is only a very rough estimate unless there is considerable historical data to support the estimated value.

Table F-5: Baseline description of the HACS Communication System at the start of requirements development (concept stage)

Task Name	Size, Eff Baseline	Cplx (*D*)
User Info Database	5,000	13
System Manager	219,000	11
Control Data Links	320,000	12
Brain	184,000	10
Sensors	157,000	12
Solitaire	74,000	12
Operating System (OS) Application Program Interface (APIs)	53,000	10
System Total	1,012,000	n/a

F.2 HACS size growth calculation

The size growth factors can be obtained from Table F-6 which has been extracted from Tables 6-3 and 6-4. The growth factors are determined by the complexity of the software functional areas (Table F-5, column 3) and from the maturity of the system being developed. The factors shown for SRR (Table F-6 forecast the mean and maximum functional area software sizes at the end of full scale development.

Table F-6: Software growth projections as a function of maturity (M) and complexity

		Complexity (*D*)							
Maturity	**M**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
SRR (Mean)	52	1.26	1.24	1.21	1.19	1.16	1.13	1.11	1.09
SRR (Max.)	52	1.89	1.81	1.72	1.64	1.55	1.46	1.38	1.29

The mean growth factor for the HACS User Info Database software with a complexity value of 13 and SRR maturity is 1.13, or 13 percent. The mean size growth[76] for the User Info Database is

$$S_{mean} = 1.13 \times 5{,}000 = 5{,}650 \text{ ESLOC}$$

The maximum growth factor for the HACS User Info Database software with a complexity value of 13 and SRR maturity is 1.46 or 46 percent. The maximum growth size is

$$S_{\max imum} = 1.46 \times 5{,}000 = 7{,}300 \text{ ESLOC}$$

Corresponding results for the remaining HACS components are also shown in Table F-7.

Table F-7 assumes the same production phase for each task at the time of the estimate and reflects both mean and maximum effective size growth from the baseline effective size.

Table F-7: Baseline description of the HACS at the start of requirements development (concept stage)

Task Name	Size, Eff Baseline	Cplx (*D*)	Size, Eff Mean	Growth Factor		Size, Eff Maximum
				Mean	Max	
User Info Database	5,000	13	5,650	1.13	1.46	7,300
System Manager	219,000	11	260,610	1.19	1.64	359,160
Control Data Links	320,000	12	371,200	1.16	1.55	496,000
Brain	184,000	10	222,640	1.21	1.72	316,480
Sensors	157,000	12	182,120	1.16	1.55	243,350
Solitaire	74,000	12	85,840	1.16	1.55	114,700
OS (APIs)	53,000	10	64,130	1.21	1.72	91,160
System Total	1,012,000	n/a	1,192,190	n/a	n/a	1,628,150

[76] The mean size growth factor is an example of an "exact approximation" that crops up frequently in estimating. The information in Table F-5 is truncated to 2 decimal places and produces a size estimate that is slightly different from the result obtained from Equation F-14. *Remember that both numbers are approximations.* For simplicity, we will use the tabular information for the size estimates in the examples.

F.3 HACS effort calculation

System-level cost estimates, using either the productivity table approach or from the Electronic Systems Center (ESC) model, can quickly produce credible "ballpark" estimates. Note, *quickly* does not equate to *simply*. This case study will use both methods in parallel so the results can be compared.

The effort projection is normally computed from the mean size growth value. This value represents the 50 percent probability of completion and is the best calculation for a realistic estimate. The baseline size is considered an optimistic, or zero growth, estimate.

The ESC estimating approach requires the software subsystems be assigned a reliability level, as defined in Section 4.4.2 of this guidebook. The HACS software subsystems generally fall into the high reliability category with the exception of the Brain, which falls into the very high category and Solitaire, which is at or below the low category.

Each subsystem contains CSCIs with different ratings. Each CSCI within a subsystem is required to interface with one or more CSCIs as part of a composite system. The interaction includes both internal interfaces as well as interfaces to external systems. The number of integrating CSCIs is defined as the total number of CSCIs in the project. ESC formed three categories based on the number of integrating CSCIs and the required reliability level for their productivity analysis as shown in Table F-8.

Table F-8: Definition of complexity/reliability categories

	Integrating CSCIs		
Reliability	**0 - 6 CSCIs**	**7 - 10 CSCIs**	**> 10 CSCIs**
Very low - nominal (Moderate loss)	Category 1	Category 1	No Data
High (Major financial loss)	Category 2	Category 2	Category 3
Very high (Public safety required)	Category 2	Category 3	Category 3

The second criterion for the ESC productivity value selection is the number of CSCIs that interface with the selected CSCI. We lack the information to determine the number of CSCIs that will ultimately be part of HACS. As a precaution, it is assumed that the number of interacting CSCIs will be between 5 and 10.

Table F-9: Productivity values for military applications by category

Project Type	Productivity (SLOC/PM)	Productivity Factor (KSLOC/PM)	Productivity Range (SLOC/PM)	Standard Deviation (SLOC/PM)
All Programs	131.7	7.60	n/a	n/a
Category 1	195.7	5.10	116.9 – 260.8	49
Category 2	123.8	8.08	88 – 165.6	23.6
Category 3	69.1	14.47	40.6 – 95.2	16.5

The ESC productivity values for each of the HACS software subsystems are drawn from Table F-9. The HACS productivity values from the ESC data are shown in Table F-10 for each of the software subsystems.

Table F-10: Productivity values for the HACS at the start of requirements review (SRR) derived from the ESC database

Task Name	Size, Eff Mean Growth	Category	Productivity Factor
User Info Database	5,650	2	124
System Manager	260,610	2	124
Control Data Links	371,200	2	124
Brain	222,640	3	69
Sensors	182,120	2	124
Solitaire	85,840	1	195
OS (APIs)	64,130	2	124
System Total	1,192,190		

The effort analysis (estimate) for each of the HACS subsystems using the ESC system-level estimating approach is shown in Table F-11. This analysis includes effort estimates for both mean and maximum growth effective size estimates.

Table F-11: Comparison of the development cost with mean effective size growth and maximum size growth using the ESC productivity model

Task Name	Productivity (SLOC/PM)	Size, Eff mean	Develop (PM)	Size, Eff Maximum	Develop (PM)	Variance (percent)
User Info Database	124	5,650	45.6	7,300	58.9	29.2
System Manager	124	260,610	2,101.7	359,160	2,896.5	37.8
Control Data Links	124	371,200	2,993.5	496,000	4,000.0	33.6
Brain	69	222,640	3,226.7	316,480	4586.7	42.1
Sensors	124	182,120	1,468.7	243,350	1,962.5	33.6
Solitaire	195	85,840	440.2	114,700	588.2	33.6
OS (APIs)	124	64,130	517.2	91,160	735.2	42.2
System Total		1,192,190	10,793.6	1,628,150	14,828.0	37.4

The productivity factor approach requires that we compare each of the HACS subsystems with the 13 system types shown in Table 7-1 to provide meaningful productivity factors for these types of systems. Our system types do not match any of the 13 types directly, so we must select the closest approximation that is consistent with the assigned complexity values. For example, the HACS Brain subsystem has a complexity value of 10 and a size value greater than 250 thousand effective source lines of code (KESLOC). The Brain approximates the System/Driver category of Table 7-1. The associated productivity value is 83 ESLOC/PM. This procedure must be applied to each of the HACS subsystems.

The productivity table assumes an average development organization operating in a normal environment. We can make subjective adjustments to the factors to compensate for application experience. *D* represents the complexity value that is typical for that software type.

Table F-12: Comparison of the development cost with mean effective size growth and maximum size growth using the productivity table model

Task Name	Productivity mean (ESLOC/PM)	Size, Eff mean	Develop (PM)	Productivity max (ESLOC/PM)	Size, Eff Maximum	Develop (PM)	Variance (percent)
User Info Database	350	5,650	16.1	350	7,300	20.9	29.8
System Manager	83	260,610	3,139.9	83	359,160	4,327.2	37.8
Control Data Links	125	371,200	2,969.6	125	496,000	3,968.0	33.6
Brain	83	222,640	2,682.4	83	316,480	3,813.0	42.1
Sensors	135	182,120	1,349.0	125	243,350	1,946.8	44.3
Solitaire	330	85,840	260.1	290	114,700	395.5	52.1
OS (APIs)	109	64,130	588.3	100	91,160	911.6	55.0
System Total	114	1,192,190	11,005.4	94	1,628,150	17,329.8	57.5

The productivity values for each HACS subsystem extracted from the productivity table are shown in Table F-12, column 2. The results of the individual HACS subsystem estimates obtained from the productivity table are summarized in Table F-12 for both mean and maximum growth, assuming the project maturity for each subsystem is established at SRR. The differences in the results of the ESC-based analyses (Table F-11) and the productivity table analysis (Table F-12) are not significant except for the User Info Database. The mean growth effort estimate of 10,794 PM obtained using the ESC approach is two percent less than the 11,005 PM in Table F-12 estimated using the productivity table approach. Some differences are not surprising since the sources of the two models are very different.

F.4 HACS Reality check

The easiest way to demonstrate the mechanics of a reality check is by example. Of course, “real” historical data is not available for the hypothetical HACS project to use in a reality check. To compensate for this reality problem, we assumed that an analysis of HACS at the component level will provide a pseudo-real level of effort; that is, a more accurate effort estimate. The detailed analysis is, theoretically, more accurate than the system-level analysis because it allows the environment and the specific CSCI products to be described. For this purpose, we continue our HACS example and delve into the reality of two estimates at the system level. Table F-13 contains a detailed component-level analysis of the HACS effort estimate for the maximum size projection. This detailed analysis is used as the basis for our pseudo-reality check. The detailed estimate is reported at the system level. In reality, the Component columns will be replaced by data from a historical database.

A cursory glance at the estimate shows productivity values that are reasonable and shows effort values that are roughly proportional to size. In most cases, the development effort in Table F-13 from the component-level analysis is close to the system-level HACS estimate. One cannot expect a perfect match, but the projections should be reasonably close. The difference between the estimate and the validation data for the nominal case is not serious. The effort of the detailed validation estimated was 16,115.2 PM against the original estimate of 17,329.8 PM, a difference of only 7 percent.

Table F-13: Comparison of the worst case (95%) cost analysis of the HACS software from a detailed component-level analysis and a productivity factor (system level) analysis

			Component		Productivity Factor	
Task Name	**Size, Eff**	**Cplx (*D*)**	**Develop (PM)**	**Productivity (ESLOC/PM)**	**Develop (PM)**	**Productivity (ESLOC/PM)**
User Info Database	7,300	13	36.1	202	20.9	350
System Manager	359,160	11	3,627.9	99	4,327.2	83
Control Data Links	496,000	12	4,203.4	118	3,968.0	125
Brain	316,480	10	4,276.8	74	3,813.0	83
Sensors	243,350	12	2,339.9	104	1,946.8	125
Solitaire	114,700	12	955.8	120	395.5	290
OS (APIs)	91,160	10	675.3	135	911.6	100
System Total	1,628,150		16,115.2	101	17,329.8	94

F.5 HACS development effort allocation

The distribution of effort varies with the effective size of the system. The larger the system the more effort will be spent in system test and integration. The relative effort spent in creating the architecture increases with size as well. The rough effort distribution as a function of source size is presented in Table F-14. See Section 4.7 of this guidebook for details about the table values.

Table F-14: Total project effort distribution as a function of product size

Size (KSLOC)	**Activity**				
	Rqmts (%)	**High Level Design (%)**	**Develop (%)**	**Sys Test (%)**	**Mgt (%)**
1	4	12	77	17	11
25	4	20	67	24	13
125	7	22	63	25	15
500	8	25	56	32	19

Adapted from McConnell, 2006; Putnam and Myers, 1992; Jones, 2000; Boehm et al, 2000; Putnam and Myers, 2003; Boehm and Turner, 2004; Stutzke, 2005.

The next step in the HACS effort estimate is to determine the effort to be assigned to each of the development and integration activities. The system test effort percentage corresponds to the value *k* specified in the following equation (F-4). The total effort is computed from the relationship

$$E_{Total} = k \times E_{Develop} \qquad \text{(F-4)}$$

where E_{Total} is the effort including both full-scale development and the CSCI integration effort,

$E_{Develop}$ is the full-scale development effort, and

k = 1.21 to 1.40 (an additional 21 to 40 percent effort) depending on the anticipated system integration difficulty (more CSCIs and more data passing between CSCIs makes the integration more difficult).

The value of *k* from the highlighted columns in Table F-14 will be used to calculate the total effort for each development task. For example, using a

size of 25 KSLOC, adding the 4 percent for requirements effort and 24 percent for system test, results in a value for *k* of 1.28. The total effort is the sum of the development effort, the requirements analysis effort and the system test effort. Development effort includes high-level design, developing the software (code and test), and management efforts.

A commercial standalone software product development can entail zero system integration. A typical *k* actor is in the range of 1.21 to 1.40 for normal system integration requirements.

The information in Table F-14 is used to roughly allocate the HACS effort defined in Table F-12 to the individual software activities. For this case study, we will use the nominal (mean growth) development effort.

Note the staffing profile for each of the subsystem developments roughly matches the Rayleigh-Norden staffing profile. It can be seen in Table F-15 as an increasing level of effort through the full-scale development with a decrease after delivery of the software to system test and integration. This is normal for all successful software development projects.

Table F-15: Total project effort distribution for the nominal HACS software development

Task Name	Total (PM)	Rqmts (PM)	High-Level Design (PM)	Develop (PM)	Sys Test (PM)	Mgt (PM)
User Info Database	19.5	0.6	2.0	12.4	2.7	1.8
System Manager	4,144.7	219.8	690.8	1,978.1	785.0	471.0
Control Data Links	3.919.9	207.9	653.3	1,870.8	742.4	445.4
Brain	3.540.8	187.8	590.1	1,689.9	670.6	402.4
Sensors	1,780.7	94.4	296.8	849.9	337.2	202.4
Solitaire	314.7	10.4	31.2	200.3	44.2	28.6
OS (APIs)	711.8	23.5	70.6	453.0	100.0	64.7
System Total	14,432.1	744.4	2,334.8	7,054.4	2,682.1	1,616.3

The nominal HACS system development effort distribution is shown in Table F-15. The system total effort of 14,432.1 PM is greater than the sum of the individual development totals (11,005.4) because the requirements and system test are added to the system development total.

Remember, the system-level estimating model does not allow for the prediction, or projection, of a software development schedule. The schedule can only be produced when the system has been decomposed to the component (CSCI) level and estimated using a component-level estimating technique.

F.6 HACS maintenance effort calculation

Let's look again at the nominal (mean) estimate for the HACS illustrated in Table F-12. The system development has been completed and delivered to the customer. There are still significant errors being uncovered in the operation, but they are decreasing as the software matures. Once the system stabilizes, from an error point of view, we assume about 8 percent of the software to be modified each year of operation to allow for enhancements (corrective, adaptive, and perfective) and knowledge retention. The maintenance functions will be performed by the software developer. What is the maximum annual effort necessary to support these requirements?

Table F-13 contains the additional complexity and productivity information necessary to project the annual maintenance effort, assuming the development staff is available to support the maintenance tasks. That is, the environment and development capability will be relatively constant and productivity will not need to be adjusted for the maintenance calculations.

The maintenance effort calculations (enhancement, knowledge retention, and steady state maintenance) are made using the Equations (4-4) through (4-6) in Section 4.8 of this guidebook. The productivity figures in Table F-16 are extracted from Table F-13 for the maximum software growth condition. These productivity values are more conservative (lower) than the values that would be extrapolated from the mean growth productivity and yields a conservative maintenance effort.

Table F-16: Maximum effort analysis of the HACS Communication System from a productivity factor (system level) analysis including annual maintenance effort

			Productivity Factor		
Task Name	**Size, Eff**	**Cplx (*D*)**	**Develop (PM)**	**Prod (ESLOC/PM)**	**Maintenance (PM/YR)**
User Info Database	5,650	13	16.1	350	4.0
System Manager	260,610	11	3139.9	83	250.6
Control Data Links	371,200	12	2969.6	125	284.9
Brain	222,640	10	2682.4	83	214.7
Sensors	182,120	12	1349.0	125	139.8
Solitaire	85,840	12	260.1	290	65.9
OS (APIs)	64,130	10	588.3	100	59.0
System Total	1,192,190		11,005.4	114	1,018.9

Applying Equations (4-4) through (4-6) to each of the major subsystems yields the anticipated annual maintenance effort for the system. Note the System Manager and Brain are dominated by the Annual Change Traffic effort and the remaining subsystems are driven by the number of people needed to retain operational system knowledge. The bulk of the personnel will come from the operational knowledge staff. The two elements, singled out in the maintenance calculation, will have to add staff to assure support for the enhancements and modifications. The annual maintenance effort is projected to be approximately 1,019 person months per year of operation. The time at which the maintenance costs balance the development effort of 11,005 PM is near 10 years. Maintenance effort can match the development effort in as few as five years, depending on the program quality and change traffic. Therefore, this maintenance effort estimate is relatively modest.

An estimate is the most optimistic prediction that has a non-zero probability of coming true.
Tom DeMarco

Appendix G

Component-Level Estimate Case Study

There are two fundamental estimating processes for software development: component and system. The component estimating process is generally used after details are available for the system, such as architectural design. At this point in time, the system definition will have functional component requirements defined (Software Requirements Specifications -- SRSs) and the interfaces between components defined (Interface Control Documents -- ICDs). The developer may be known at this time, or at least the developer class will be known well enough that the typical characteristics can be defined.

The material in this appendix demonstrates developing estimates at the component level and is supported by the same case study introduced in Appendix F. Evolving fragments of the estimate will be updated as each step, or portion of a step, is completed. The Hierarchical Autonomous Communication System (HACS) system contains several components (CSCIs).

G.1 HACS baseline size estimate

HACS has been created or, rather, translated as a demonstration vehicle for the cost estimation process. The system contains several subsystems that have been refined to the component (CSCI) level. The component size information at the start of the estimate is shown in Table G-1.

Table G-1: Baseline description of the HACS Communication System at the start of requirements review (SRR)

Task Name	Size, Eff Baseline	Cplx	Units
User Info Database (DB)	n/a	13	FP
Glue code for DB	n/a	13	FP
DB Configuration	n/a	0	FP
System Manager	219,000	11	ESLOC
Control Data Links	320,000	12	ESLOC
Satellite	152,000	12	ESLOC
Transmitter	96,000	12	ESLOC
Receiver	56,000	12	ESLOC
Radio	84,000	12	ESLOC
Fiber link	84,000	12	ESLOC
Brain	184,000	10	ESLOC
Sensors	157,000	12	ESLOC
Solitaire	74,000	12	ESLOC
Operating System (OS) Application Program Interfaces (APIs)	53,000	10	ESLOC
System Total	1,012,000		ESLOC

The Cost Analysis Requirements Description (CARD) specifies the Control Data Links, with a forecast baseline size of 320,000 effective source lines of code (ESLOC), and the User Info Database, specified in function points (FPs). Since the effective size of the Control Data Links subsystem is too large to be developed as a single CSCI (i.e., > 200,000 effective source lines of code [ESLOC]), a realistic component-level estimate cannot be completed without further decomposition. Some estimating tools, however, allow a CSCI to be greater than 200,000 ESLOC, but the estimate results will be questionable since insufficient historical data exists (little to none) to support the tools in this size region. Also, since FPs are involved in the system size specification, they must be converted to ESLOC before the estimate can proceed[77].

CARD specifications place the HACS software subsystems generally into the High reliability category

[77] The information regarding the conversion of function points to SLOC in this section is essentially a duplicate of the conversion described in Section 6.

with the exception of the Brain, which falls into the Very High category, and Solitaire, which is at or below the Low category.

The system-level estimate of the HACS example in Appendix F assumed the major subsystems were defined at only the system level. From that level, a "ballpark" estimate for the entire HACS software product was constructed. It was impossible to create a component level estimate because several of the subsystems were above the "magic" 200,000 ESLOC CSCI size limit. In this estimate, the components (CSCIs) have been defined, and a more refined and accurate estimate can be constructed for the development.

The expanded portion (to the CSCI level) is shown in the white and yellow areas of Table G-1. The nominal analysis assumes the estimate is being made at the time of the Software Requirements Review (SRR) and effective size reflects the baseline level of software growth. For example, the baseline size of the Brain software contains 184,000 ESLOC. Note the Brain baseline size is near the 200,000 ESLOC limit and can create problems as the estimate progresses if growth and/or risk are involved.

G.2 HACS size estimate

Size is the most important effort (cost) and schedule driver in any software development estimate. Size information early in a project is seldom, if ever, accurate. Information presented in a CARD is generally predicted during a concept development phase prior to the start of full-scale development (pre-Milestone B). Size information is often represented by a single value which leads to a point cost and schedule estimate. However, the information may be more accurately expressed as three values representing a minimum (most optimistic), a most likely, and a maximum (most pessimistic), to give a more realistic picture of the software size information.

The HACS User Info Database software is initially specified as FPs for the development estimate. The unadjusted function point (UFP) count is shown in Table G-2. Since FPs are involved in the system size specification, they must be converted to SLOC before the estimate can proceed to the ultimate specification of the system size. The FP values in the table have been derived from the HACS specification. The UFP count for the Info Database task is 84.

Table G-2: HACS Info Database unadjusted function point calculation

Component Types		Ranking		Total
	Low	Average	High	
Internal Logical File	1	1	0	17
External Interface File	1	3	0	26
External Input	2	3	0	18
External Output	0	3	0	15
External Inquiry	0	2	0	8
Transform	0	0	0	0
Transition		0		0
Unadjusted Function Points				**84**

The next step in the equivalent baseline software size calculation for the HACS User Info Database is to adjust the FP size to account for differences

between the database characteristics and the normal characteristics assumed by the UFP count. This involves adjusting the UFP count by the General System Characteristics (GSCs) Value Adjustment Factor (VAF) rating. Once the 14 GSCs for the database have been evaluated, as demonstrated for HACS in Table G-3, the value adjustment factor can be computed using the IFPUG VAF equation. The equation is

$$VAF = 0.65 + \left[\left(\sum_{i=1}^{14} GSC_i \right) / 100 \right] \qquad \text{(G-1)}$$

where GSC_i = degree of influence for each GSC, and
i = 1 to 14 representing each GSC.

Table G-3: HACS GSC ratings

No.	General Characteristic	Value	No.	General Characteristic	Value
1	Data communications	0	8	Online update	5
2	Distributed data processing	4	9	Complex processing	0
3	Performance	3	10	Reusability	0
4	Heavily used configuration	4	11	Installation ease	4
5	Transaction rate	5	12	Operational ease	5
6	Online data entry	5	13	Multiple sites	4
7	End-user efficiency	0	14	Facilitate change	4

The resulting score will be between 0.65 and 1.35, depending on the sum of the individual ratings. The sum of the GSCs for the database is

$$\sum_{i=1}^{14} GSC_i = 43$$

The *VAF* result for the HACS Info Database is then

$$VAF = 1.08$$

The adjusted function point (AFP) count is obtained by multiplying the VAF times the UFP count. The standard AFP count is given by

$$AFP = VAF \times UFP \qquad \text{(G-2)}$$

The AFP count for the HACS Info Database is

$$AFP = 1.08 \text{ x } 84 = 91 \text{ FPs.}$$

The conversion of FPs (UFP or AFP) to equivalent total SLOC is accomplished by multiplying AFP by a scaling factor (backfire factor [BF]) to produce the total size as

$$S_t = AFP \times BF \qquad \text{(G-3)}$$

where S_t = total software SLOC count,
AFP = adjusted FP count, and
BF = SLOC/FP backfire factor from Table 6-17

or,

1.08 x 84 x 55 = 5,000 (approximately) total SLOC.

The resulting User Info Database effective size for the HACS software is presented in Table G-4 assuming the implementation language for the User Info Database is C++ and *BF = 55*. Note: the resulting source code value is for baseline total source code. There is no pre-existing code assumed in this value, and the computed equivalent size does not assume any size growth during the project development.

Table G-4: Baseline description of the HACS Communication System at the start of requirements review (SRR)

Task Name	Size, Eff Baseline	Cplx	Units
User Info Database			
Glue code for DB	5,000	13	Function Points
DB Configuration			
System Manager	219,000	11	ESLOC
Control Data Links	320,000	12	ESLOC
Satellite	152,000	12	ESLOC
Transmitter	96,000	12	ESLOC
Receiver	56,000	12	ESLOC
Radio	84,000	12	ESLOC
Fiber link	84,000	12	ESLOC
Brain	184,000	10	ESLOC
Sensors	157,000	12	ESLOC
Solitaire	74,000	12	ESLOC
OS (APIs)	53,000	10	ESLOC
System Total	1,012,000		ESLOC

The baseline size is the starting size for the effort analysis. The baseline values from the CARD are summarized in Table G-4, column 2. These values assume no software growth or difficulty during development. At the concept level, the size is only a very rough estimate unless there is considerable historic data to support the value.

G.3 HACS size growth calculation

The software requirements have been developed and the size projections have been somewhat refined at this point in time. There has already been some size growth since the concept phase of the development – this is included in the current baseline. Our task is to project the growth from SRR to delivery of the HACS product. The size information shown in Table G-4, columns 2 and 3, represents the baseline size and complexity projected at the beginning of SRR.

The size growth factors can be obtained from Table G-5 which has been extracted from Tables 6-3 and 6-4. The growth factors are determined by the complexity of the software product and from the maturity of the product at the time the estimate is being made. The factors project the mean and maximum product sizes at the end of development. The rows in Table G-5 correspond to a project maturity at the start of SRR.

Table G-5: Software growth projections as a function of maturity (M) and complexity.

		Complexity							
Maturity	**M**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
SRR (Mean)	52	1.26	1.24	1.21	1.19	1.16	1.13	1.11	1.09
SRR (Max)	52	1.89	1.81	1.72	1.64	1.55	1.46	1.38	1.29

The mean growth factor for the HACS User Info Database software with a complexity value of 13 and SRR maturity is 1.13 or 13 percent. The mean size growth factor[78] is

$$S_{mean} = 1.13 \times 5000 = 5{,}650 \quad \text{ESLOC} \qquad \text{(G-4)}$$

The maximum growth factor for the HACS User Info Database software with a complexity value of 13 and SRR maturity is 1.46, or 46 percent. The maximum growth size is

$$S_{\max imum} = 1.46 \times 5000 = 7300 \quad \text{ESLOC} \qquad \text{(G-5)}$$

Corresponding results for the remaining HACS components are shown in Table G-6. The table assumes all components are in the same production phase at the time of the estimate. The mean and maximum effective size growth is computed from the baseline effective size.

Table G-6: Baseline description of the HACS Communication System at the start of requirements development

Task Name	Size, Eff Baseline	Cplx	Size, Eff Nominal	Growth Factor		Size, Eff Maximum
				Nom	Max	
User Info Database	5,000		5,650			7,300
Glue code for DB	5,000	13	5,650	1.13	1.46	7,300
DB configuration						
System Manager	219,000	11	260,610	1.19	1.64	359,160
Control Data Links	320,000	12	371,200			496,000
Satellite	152,000	12	176,320			223,201
Transmitter	96,000	12	111,360	1.16	1.55	140,969
Receiver	56,000	12	64,960	1.16	1.55	82,232
Radio	84,000	12	97,440	1.16	1.55	123,348
Fiber link	84,000	12	97,440	1.16	1.55	123,348
Brain	184,000	10	222,640	1.21	1.72	316,480
Sensors	157,000	12	182,120	1.16	1.55	243,350
Solitaire	74,000	12	85,840	1.16	1.55	114,700
OS (APIs)	53,000	10	64,130	1.21	1.72	91,160
System Total	1,012,000		1,192,190			1,628,150

The effort and schedule projections are normally computed from the mean size growth value. This is the best calculation for a realistic estimate. The probability of the baseline size existing at the end of development is very small and the delivery schedule and development effort are unlikely.

[78] The mean size growth factor is an example of an "exact approximation" that crops up frequently in estimating. The information in Table G-5 is truncated to two decimal places and produces a size estimate that is slightly different from the result obtained from Equation 6-14. *Remember that both numbers are approximations.* For simplicity we will use the tabular information for the size estimates in the examples.

The nominal effective size that will be used in the HACS cost and schedule calculations corresponds to the mean product size growth. The baseline size is considered an optimistic, or zero growth, estimate.

G.4 HACS environment

The next step in the HACS component-level estimate is an evaluation of the software environment. This evaluation is conducted in two parts: (1) an estimate of the developer capability of each subsystem, or in the case that a single contractor (developer) may be responsible for multiple subsystems, an estimate of each developer's capability, and (2) an estimate of the remaining environment factors for each CSCI that are impacted by the HACS product development. Each CSCI can have a unique set of experience, development system, management, and product factors.

G.4.1 HACS developer capability

The HACS development organization is assumed to employ an average development staff located in a facility with a typical cubical arrangement. The application experience of the development team is an average of 3 to 5 years in systems similar to HACS. The CMMI rating is Level 2 and preparing for a Level 3 evaluation. The hardcopy turnaround time, a measure for computing resource access, is less than 30 minutes. Terminal response time is approximately 500 milliseconds. The series of calculations—outlined earlier in Section 5.4.4, Equation (5-4)—yields a basic technology constant for the HACS software development organization. The HACS basic technology rating results are shown in Table G-7.

Table G-7: Parameter values for basic capability estimate calculation

Parameter	Value
Analyst capability – ACAP	1.00
Application domain experience – AEXP	0.89
Programmer capability – PCAP	1.00
Use of modern practices – MODP	1.04
Automated tool support – TOOL	1.03
Hardcopy turnaround time – TURN	0.93
Terminal response time – RESP	1.00
Basic Technology Constant	6474

G.4.2 Personnel evaluation

The personnel in the HACS project, shown in Table G-8, are assumed to be trained in the development system and associated practices. A value of 1.0 indicates no impact on the development. There is an average experience level of slightly less than two years using the development system. There are no changes in development practices planned during the development; hence, there is no impact. Process improvement would increase the practices experience (PEXP) value if the development organization imposed development practices on the project. The process changes will not impact this project. The programmers are masters of the required programming language. In this example, the language is C++.

Table G-8: Personnel parameter values for HACS estimate

Parameter	Value
Development system experience - DEXP	1.01
Programming language experience - LEXP	1.00
Development practices experience – PEXP	1.00
Target system experience - TEXP	1.00
Multiple organizations - MORG	1.10
Multiple development sites - MULT	1.07

The HACS management factors show actions that will have an impact on productivity. The development organization will be using personnel from other organizations, and the software development will be accomplished with the development people spread across multiple sites in close proximity. The combined impact (product) of these parameters (MORG and MULT) amounts to approximately an 18 percent productivity penalty.

G.4.3 Development environment

The HACS development environment is the third project environment area of interest. The environment parameters determine the impact of the stability and availability of the environment; that is, the volatility of the development system and associated practices, the proximity of the resource support personnel, and access to the development system.

The HACS development practices are expected to experience minor changes on a monthly average during the development due to process improvement. Process improvement should ultimately improve productivity, but will decrease productivity in the short term until the new practices stabilize. The development system will show productivity loss during system updates that will also occur on a monthly average. These are normal conditions. The resource and support personnel are not located in the software development areas, but are within 30 minutes in case of system or compiler problems. System access is 100 percent of the first shift workday, so there will be no impact due to limited access. The HACS development environment parameter values are shown in Table G-9.

Table G-9: Development environment parameter values for HACS estimate

Parameter	Value
Practices volatility - PVOL	1.08
Development support location - RLOC	1.08
Development system volatility - DVOL	1.07
Development system dedication - RDED	1.00

The combined impact on development productivity of this parameter group is approximately 25 percent beyond the ideal environment. The individual parameter ratings are described in more detail in Section 8 of this guidebook.

G.4.4 HACS product impact

The fourth environment factor group is used to evaluate the impact of the product characteristics and constraints on the project productivity. The information in this group is reasonably constant for a given product type, and can be collected in a template that may have only a few changes within the product type due to specific product implementations. Within the HACS system, as shown in Table G-10, there are several CSCI types that could be defined as multiple product templates for the effort and schedule analysis.

Table G-10: Product impact parameter values for HACS estimate

Parameter	Value
Multiple security classification - MCLS	1.00
Special display requirements - DISP	1.05
Dev. To Target system rehost - HOST	1.00
Target memory constraints - MEMC	1.00
Product quality requirements - QUAL	1.03
Real-time requirements - RTIM	1.00
Requirements volatility - RVOL	1.15
Software security requirements - SECR	1.00 - 1.34
Product development standards - SPEC	1.21
Product test requirements - TEST	1.05
Product timing constraints - TIMC	1.00
Target system volatility - TVOL	1.00

The HACS project requires all development personnel to have the same security clearance level so there is no productivity loss due to the personnel working across security boundaries. The system requires a user-friendly interface which results in more complicated design and test activities; thus, a penalty of 5 percent is specified. There are no hardware re-hosting requirements. The product is not memory constrained and there are no real-time requirements or timing constraints. The Common Criteria Evaluation Assurance Level (CC EAL) rating[79] for the product is Level 1 except for the Brain (a complex system/driver or process control function), which has a CC EAL rating of Level 4. The target system is in common use and is stable. The required reliability for the development is satisfied by following the ISO/IEC 12207

[79] The CC for Information Technology Security Evaluation is an international standard (ISO/IEC 15408) for computer security.

standard. The penalties corresponding to the ISO standard are specified in the QUAL, SPEC and TEST parameters.

The requirements volatility definition we will follow for the HACS software is "known product with occasional moderate redirection," since the product will have some functionality improvements beyond the existing products.

The combined productivity impact beyond the ideal environment is 58 percent for the bulk of the system, but 112 percent for the Brain. The penalties may seem high at first glance, but the change is from the ideal which does not occur in practice. The individual parameter ratings are described in more detail in Section 8 of this guidebook.

G.4.5 HACS effective technology constant

The productivity factor for Jensen-based models is the effective technology constant C_{te} defined in more detail in Section 5.4.5, Equation (5-5). This equation combines the basic technology constant with the product of the development environment factors f_i. The resulting productivity factor is

$$C_{te} = \frac{C_{tb}}{\prod_i f_i} \qquad \text{(G-6)}$$

The median productivity factor C_{te} for the HACS software development is computed to be

$$C_{te} = \frac{6{,}474}{(1.58 \times 1.25 \times 1.18)} = 3{,}150 \qquad \text{(G-7)}$$

The mean value, considering the worst case value of 3,150, is shown in Table G-11. The effective technology constant for the Brain is 2,351, a 34 percent drop compared to the bulk of the system. This is due to the EAL security rating of Level 4. The HACS technology constant values are shown in Table G-11.

Table G-11: Technology constant values for HACS estimate

Parameter	Value
Basic technology constant - C_{tb}	6,474
Effective technology constant - C_{te}	3,150 / 2,351

A 2:1 ratio of the basic and effective technology constants is common. It is not uncommon for ratios of 4:1 to exist for highly constrained or secure software systems.

G.5 HACS development effort and schedule calculations

The cost analysis of the HACS software is conducted at the component (CSCI) level and rolled up into the next higher level. For example, the Transmit and Receive components are estimated and rolled up into the Satellite super component. The Radio and Fiber Link components are estimated and rolled up with the Satellite into the Control Data Links subsystem. Each of the components is within the range of 5,000 to 200,000 ESLOC (except for the System Manger and the Brain) and compatible with the limitations of the estimating models and reality. Some estimating models allow component sizes greater than 200,000 ESLOC, but reality places a limit of 200,000 ESLOC on deliverable software components.

The result of the HACS estimate under mean software growth conditions is presented in Table G-12. The worst case growth estimate is contained in

Table G-13. The calculations for these estimates were performed by the Sage 3 estimating tool which is based on the Jensen II model.

Table G-12: Nominal effort and schedule (in months) analysis of the HACS at the component (CSCI) level

Task Name	C_{tb}	C_{te}	Size, Eff	Cplx	Develop (PM)	Total Effort (PM)	Prod (ESLOC/PM)	Dev Sched (mo)	Total Sched (mo)
User Info Database			5650	13	30.7	46.6	212.6	9.0	12.0
Glue code for DB	6416	3150	5650	13	26.7	38.6	212.6	9.0	12.0
DB Configuration					4.0	8.0		3.0	4.0
System Manager	6416	3150	260,610	11	2455.9	3544.2	105.8	43.4	57.7
Control Data Links			371,200	12	2,973.2	4290.6	125.5	30.4	40.4
Satellite	6416	3150	176,320	12	1403.6	2025.6	127.9	30.4	40.4
Transmit	6416	3150	111,360	12	921.2	1329.4	121	30.4	40.4
Receive	6416	3150	64,960	12	482.4	696.2	134.8	24.5	32.6
Radio	6416	3150	97,440	12	784.8	1132.5	124.3	28.8	38.3
Fiber Link	6416	3150	97,440	12	784.8	1132.5	124.3	28.8	38.3
Brain	6416	2351	222,640	10	2795.0	4033.6	79.8	46.8	62.2
Sensors	6416	3150	182,120	12	1622.2	2398.8	109.7	37	49.2
Solitaire	6416	3150	85,840	12	674.0	972.7	127.5	27.4	36.4
OS (APIs)	6416	3150	64,130	10	441.8	637.6	145.3	25.3	33.6
System Total			1,192,190		10,992.8	15,924.1	106.6	43.4	57.7

Table G-13: Worst-case effort and schedule analysis of the HACS at the component (CSCI) level

Task Name	Ctb	Cte	Size, Eff	Cplx	Develop (PM)	Total Effort (PM)	Prod (ESLOC/PM)	Dev Sched (mo)	Total Sched (mo)
User Info Database			7321	13	40.2	60.3	202	13.1	17.4
Glue code for DB	6474	3150	7321	13	36.2	52.3	202	10.1	13.4
DB Configuration					4.0	8.0		3.0	4.0
System Manager	6474	3150	358,309	11	3612.0	5212.6	99.2	49.3	65.6
Control Data Links			496,035	12	4205.3	31,854.8	118.4	34.1	45.4
Satellite	6474	3150	235,617	12	1985.3	2865.1	120.7	34.1	45.4
Transmit	6474	3150	148,811	12	1302.9	1880.3	114.2	34.1	45.4
Receive	6474	3150	86,806	12	682.4	984.8	127.2	27.5	36.6
Radio	6474	3150	130,209	12	1110.0	1601.9	117.3	32.3	43
Fiber Link	6474	3150	130,209	12	1110.0	1601.9	117.3	32.3	43
Brain	6474	3150	316,869	10	4262.5	6151.4	74.3	53.8	71.6
Sensors	6474	3150	243,368	12	2351.1	3393.0	103.5	41.5	55.2
Solitaire	6474	3150	114,708	12	953.4	1375.9	120.3	30.8	40.9
OS (APIs)	6474	3150	91,272	10	673.7	972.3	135.5	29.1	38.7
System Total			1,627,882		16,094.2	47,198.5	101.2	49.3	65.6

There are two important items to note in Tables G-12 and G-13. First, a fixed effort is added to the User Info Database CSCI that is not estimated using the system- or component-level approaches described in Sections 4 and 5. The addition is included to account for the effort required to configure the database.

Second, nominal (Table G-12) Brain component size is 222,640 ESLOC, which violates the reality size constraint and has an estimated schedule of 62.2 months, which violates the reality schedule constraint. Table G-13 Brain size is 316,869 ESLOC, which has a very small probability of success if built as identified based on historical data. The estimate of this component is outside the data upon which the estimating models and tools have been built.

G.6 Verify HACS estimate realism

The easiest way to demonstrate the mechanics of a reality check is by example. Table G-12 contains the HACS effort and schedule estimate at the component level for the most likely size projection. Since there is no historical data available for the reality check, we will use the Productivity Factor Table (from the earlier Section 4.4.1, Table 4-5), which is based on a collection of historical data. The two components of the satellite subsystem within the Control Data Links are typical telecommunication systems. Without interpolating between columns in the table, the nearest size column is 100K ESLOC. The Transmit component has an estimated productivity of 114 ESLOC/PM and the Receive component productivity is 127 ESLOC/PM. The table reference productivity of 100 ESLOC/PM is within 27 percent of the two estimated productivity values. This error is not excessive considering that the values in the table do not reflect the developer or the development environment for the HACS project. The productivity table does not allow any comparison of the estimated schedules for reality check purposes.

A cursory glance at the estimate shows nominal productivity and effort values which are roughly proportional to size. However, the schedule values are inconsistent; that is, the longest and shortest schedules are not proportional.

In most cases, the effort in Table G-12 does not disagree greatly with the system level HACS estimate using the productivity factor data from Table 4-5. One cannot expect a perfect match, but the projections should be reasonably close.

The System Manager and the brain are larger than the 200K ESLOC limit assuming mean growth. At this point in the development, this can occur due to two basic reasons. First, the detailed requirements allocations have not been identified and therefore we are creating an effort (cost) allocation for the subsystem. The second reason is that there are classified components in this subsystem and to keep the document unclassified, the requirements are not detailed. Care must be taken in either case that these CSCIs are carefully tracked to ensure that they do not derail the project.

CSCIs with an effective size larger than 200,000 ESLOC are likely to be undeliverable although there are exceptions to every rule. The HACS system as a whole fits the telecommunication category task better than any other. We must apply judgment to the subsystems in the HACS system using the modified McConnell data. For example, the Brain task is most like

an embedded system but is beyond the 200K ESLOC boundary. If we extrapolate the Brain productivity to the 250K ESLOC column, we would find that the estimated productivity of 74 ESLOC/PM is within 11 percent of the 83 ESLOC/PM entry in Table 7-1.

The Brain and the System Manager CSCIs exceed the 200 KESLOC size limit. Persisting with the current composition of the Brain and System Manager are likely to create undeliverable products. The schedule for the Brain as an example, assuming it is a single CSCI, is almost 72 months, or six years. Again, the schedule indicates a non-deliverable product because we look for a schedule of five years or less. The Brain can (must) be decomposed into smaller CSCIs which will, in turn, decrease the required schedule by developing it in smaller components. The smaller components will achieve higher productivity, which will improve the development effort and delivery potential.

The Brain is a candidate for previously created and CC EAL certified software (either GOTS or COTS) which could effectively reduce the size of the subsystem.

A second part of any reasonable estimate is the consideration of risk. Software risk often presents itself as an effective size increase. Table G-12 shows the HACS system estimate assuming mean size growth. Table G-13 shows the worst case size or risk projection.

G.7 Allocate HACS development effort and schedule

G.7.1 HACS effort allocation

The distribution of effort varies with the effective size of the system. The larger the system the more effort will be spent in system test and integration. The relative effort spent in creating the architecture increases with size as well. The rough effort distribution as a function of source size is presented

Table G-14: Total project effort distribution for the HACS software development

Task Name	Activity				
	Rqmts (%)	High-Level Design (%)	Develop (%)	Sys Test (%)	Mgt (%)
User Info Database	4	10	64	17	9
System Manager	8	17	38	32	13
Control Data Links	8	17	38	32	13
Brain	8	17	38	32	13
Sensors	8	17	38	32	13
Solitaire	7	16	48	25	11
OS (APIs)	7	16	48	25	11

in Table G-14.

The next step in the HACS effort estimate is to assign the effort to each of the development and integration activities. If system integration is to be included in the total system development, the effort computed in the last step must be increased to account for the total system development cost.

The total effort is computed from the relationship

$$E_{Total} = k \times E_{Develop} \qquad \text{(G-8)}$$

where E_{Total} is the effort including both full-scale development and the CSCI integration effort,

$E_{Develop}$ is the full-scale development effort, and

k = 1.21 to 1.32 depending on the anticipated system integration difficulty.

The value of k from the highlighted columns in Table G-14 will be used to calculate the total effort for each development task. For example, using a size of 25 KESLOC, adding the 4 percent for requirements effort and 24 percent for system test results in a value for *k* of 1.28. The total effort is the sum of the development effort, the requirements analysis effort and the system test effort. Development effort includes high-level design, development, and management efforts.

A commercial standalone software product development can entail zero system integration. A typical *k* factor is in the range of 1.21 to 1.32 for normal system integration requirements. In this example, the value of *k* from the highlighted columns in Table G-14 will be used to calculate the total effort for each development task.

The resulting effort allocation for the mean growth (nominal) condition is summarized in Table G-15. The effort allocation process for the baseline and maximum growth conditions are computed in the same fashion as the mean growth condition.

Table G-15: Nominal effort allocation for the HACS Communication System at the component (CSCI) level

Task Name	Size, Eff (Nominal ESLOC)	Total Effort (pm)	Rqmts (pm)	High-Level Design (pm)	Develop (pm)	Sys Test (pm)	Mgt (pm)
User Info Database	5650	46.6	1.5	3.9	32.7	6.6	3.5
Glue code for DB	5650	38.6	1.5	3.9	24.7	6.6	3.5
DB Configuration		8.0			8.0		
System Manager	259,804	3544.2	283.5	602.5	1,346.8	1,134.1	460.7
Control Data Links	371,561	4,290.6	300.4	686.5	2,059.5	1,072.6	472.0
Satellite	176,491	2025.6	141.8	324.1	972.3	506.4	222.8
Transmit	111,468	1329.4	93.1	212.7	638.1	332.3	146.2
Receive	65,023	696.2	48.7	111.4	334.2	174.1	76.6
Radio	97,535	1132.5	79.3	181.2	543.6	283.1	124.6
Fiber Link	97,535	1132.5	79.3	181.2	543.6	283.1	124.6
Brain	222,917	4033.6	322.7	685.7	1,532.8	1,290.8	524.4
Sensors	182,297	2398.8	167.9	383.8	1,151.4	599.7	263.9
Solitaire	85,923	972.7	68.1	155.6	466.9	243.2	107.0
OS (APIs)	64,210	637.6	44.6	102.0	306.0	159.4	70.1
System Total	1,192,392	15,924.1	1,199.5	2,643.8	6,655.9	4,674.4	1,949.5

Note that the effort allocation for the Brain and the System Manager are suspect because the total effort calculation is derived from a faulty assumption that the components are being developed as single CSCIs. The sizes are beyond the 200,000 ESLOC limit for single CSCIs. The Brain and System Manager each will likely be decomposed into at least two CSCIs before the start of development. If they are not decomposed, they become high risk developments.

The effect of CSCI complexity can be seen in Table G-15 as an increasing level of effort through the full-scale development with a decrease after delivery of the software to system test and integration. The total effort for each task within the HACS project is incorporated in the total effort requirements for the nominal HACS analyses. The total effort is computed from the information in the project effort distribution table (Table G-14).

G.7.2 Schedule allocation

Schedule allocation depends on the effective size of the software system AND the software development approach. A classic waterfall development will meet schedule milestones at different times than an incremental or an agile development. There are some guidelines that serve to place approximate milestones for ballpark estimates. The broad schedule breakdown is shown in Table G-16. Allocating milestones in the program plan requires the number ranges to be reduced, again depending on the development approach.

Using the schedule breakdown table described in Section 5.7.2, we see that the baseline and mean and maximum growth sizes for the command and control project are all close enough to the 125 KESLOC entry in that we can apply the percentage allocation for each activity directly from the table. The table is repeated here (Table G-16) for reference.

The overall schedule distribution for a project is based on the entire process totaling 100 percent of the schedule. Judgment is necessary when selecting the specific values from the table for the project in question. The percentages in Table G-16 for the HACS project are enclosed in parentheses.

Table G-16: Approximate schedule breakdown as a function of product size

Size (KSLOC)	Activity			
	Requirements (Percent)	High-Level Design (Percent)	Development (Percent)	System Test (Percent)
1	6-16 (6)	15-25 (20)	50-65 (55)	15-20 (19)
25	7-20 (7)	15-30 (19)	50-60 (52)	20-25 (22)
125	8-22 (8)	15-35 (18)	45-55 (49)	20-30 (25)
500	12-30 (12)	15-40 (15)	40-55 (43)	20-35 (30)

Sources: Adapted from McConnell, 2006; Putnam and Myers, 1992; Boehm et al, 2000; Putnam and Myers, 2003; Stutzke, 2005

The schedule allocation for the HACS project is presented in Table G-17. As with the effort allocation, the schedule allocation for both the Brain and

System Manager are also suspect because of the extremely high effective source size for the CSCI. The schedule allocation in Table G-17 can be used as a conservative schedule estimate because the CSCI will undoubtedly be decomposed into smaller CSCIs that will require a smaller total schedule. The Brain and System Manager schedules of almost six years, assuming worst-case growth, makes the CSCIs high-risk components that will likely never be delivered in their present level of decomposition.

The HACS plan in Table G-17 assumes all tasks are started at completion of the requirements review; phasing is not accounted for in the table. A dependency of one task on another task's completion is not accounted for in this table. In practice, each task is independently programmed on a master schedule for planning purposes.

Table G-17: Nominal schedule allocation for the HACS at the component (CSCI) level

Task Name	Size, Eff (Nominal ESLOC)	Total Schedule (mo)	Rqmts (mo)	High-Level Design (mo)	Develop (mo)	Sys Test (mo)
User Info Database	5680	12.1	0.7	2.4	6.7	2.3
Glue code for DB	5680	12.1	0.7	2.4	6.7	2.3
DB Configuration						
System Manager	259,804	57.7	5.8	10.4	26.0	15.6
Control Data Links	371,561	40.4	3.2	7.3	19.8	10.1
Satellite	176,491	40.4	3.2	7.3	19.8	10.1
Transmit	111,468	40.4	3.2	7.3	19.8	10.1
Receive	65,023	32.6	2.6	5.9	16.0	8.2
Radio	97,535	38.3	3.1	6.9	18.8	9.6
Fiber Link	97,535	38.3	3.1	6.9	18.8	9.6
Brain	222,917	62.2	6.2	11.2	28.0	16.8
Sensors	182,297	49.2	4.9	8.9	22.1	13.3
Solitaire	85,923	36.4	2.6	6.9	18.9	8.0
OS (APIs)	64,210	33.6	2.4	6.4	17.5	7.4
System Total	1,192,392	57.7	6.2	11.2	28	16.8

G.8 Allocate HACS maintenance effort

Let's look again at the nominal estimate for the HACS Communication System as illustrated in Table G-12. The system development has been completed and delivered to the customer. There are still significant errors being uncovered in operation, but they are decreasing as the software matures. Once the system stabilizes from an error point of view, we expect about 8 percent of the software to be modified each year of operation to allow for enhancements and operational refinements. The maintenance functions will be performed by the software developer. What is the annual effort necessary to support these requirements?

Table G-12 contains the complexity information necessary to project the annual maintenance effort, assuming the development staff is available to

support the maintenance tasks. That is, the environment and development capability will be relatively constant and productivity will not need to be adjusted for the maintenance calculations.

The maintenance (enhancements, knowledge retention, and steady state maintenance) effort calculations are made using the Equations (4-4) through (4-6) in Section 4.8. The productivity figures in Table G-18 are extracted from Table G-13 for the maximum software growth condition. These productivity values are more conservative (lower) than the values that would be extrapolated from the mean growth productivity and yield a conservative maintenance effort.

Table G-18: Nominal component (CSCI) level cost analysis of the HACS Communication System inc. maintenance

Task Name	Ctb	Cte	Size, Eff	Cplx	Develop (PM)	Total Effort (PM)	Prod. (ESLOC/PM)	Maintenance Effort (PM/year)
User Info Database			5680	13	30.7	46.6	202	4.0
Glue code for DB	6474	3150	5680	13	26.7	38.6	202	4.0
DB Configuration					4.0	8.0		
System Manager	6474	3150	259,804	11	2455.9	3544.2	99.2	217.3
Control Data Links			371,561	12	2,973.2	4290.6	118.4	276.0
Satellite	6474	3150	176,491	12	1403.6	2025.6	120.7	126.4
Transmit	6474	3150	111,468	12	921.2	1329.4	114.2	85.5
Receive	6474	3150	65,023	12	482.4	696.2	127.2	40.9
Radio	6474	3150	97,535	12	784.8	1132.5	117.3	74.8
Fiber Link	6474	3150	97,535	12	784.8	1132.5	117.3	74.8
Brain	6474	2351	222,917	10	2795.0	4033.6	74.3	205.1
Sensors	6474	3150	182,297	12	1622.2	2398.8	103.5	140.9
Solitaire	6474	3150	85,923	12	674.0	972.7	120.3	65.9
OS (APIs)	6474	3150	64,210	10	441.8	637.6	135.5	59.1
System Total			1,192,392		10,992.8	15,924.1	101.2	968.3

Applying Equation (4-6) to each of the major subsystems yields the anticipated annual maintenance effort for the system. Note the System Manager and Brain are dominated by the Annual Change Traffic (ACT) effort and the remaining subsystems are driven by the number of people needed to retain system knowledge. The bulk of the personnel will come from the operational knowledge staff. The two elements singled out in the maintenance calculation will need to add staff to insure support for the enhancements and modifications. The annual maintenance effort is projected to be approximately 968 PM per year of operation. The time at which the maintenance costs balance the development effort of 11,993 PM is nearly 11 years. Maintenance effort can match the development effort in as few as five years, depending on the program quality and change traffic. Therefore, this maintenance effort is relatively modest.

The annual maintenance effort is projected to be approximately 1,019 PM per year of operation according to the system-level estimate in Table F-15. The annual maintenance effort at the nominal component level is 968.3 PM per year, which is less than the system-level maintenance estimate by 5

percent. As a rough order of magnitude test, the maintenance level matches the development effort at about 10 years.

Make everything as simple as possible, but not simpler.

Albert Einstein

Appendix H

The Defense Acquisition System

The warfighter relies on discipline, courage, training, and superior equipment to succeed. The purpose of the acquisition process is, ultimately, to support the warfighter. The Defense Acquisition System exists to manage the Nation's investments in technologies, programs, and product support necessary to achieve the National Security Strategy and support the United States Armed Forces.[80] The acquisition system consists of the policies and procedures governing the operations of the entire Department of Defense (DoD) acquisition process. As described in Figure H-1, the system is designed around a series of life-cycle phases. As the process moves through these phases, decision points determine the next set of tasks.

Figure H-1: Defense Acquisition System

The acquisition process, referred to as the Defense Acquisition Management Framework[81], is a continuum of activities that represent or describe acquisition programs. It is intended to provide faster delivery of capabilities or improvements to the warfighter through a simple, flexible approach. Acquisition management policies aim to reduce total ownership costs "cradle-to-grave" by addressing interoperability, supportability, and affordability.

Cost estimates are one of the fundamental building blocks of the acquisition process. The cost estimate and supporting budget are a part of the baseline cost and schedule against which a program's progress and success are measured. The government must be able to produce and use cost estimates in order to evaluate whether a system is affordable and consistent with both the DoD components' and their overall long-range investment and force structure plans. Cost estimates also form the basis for budget requests to Congress.

> The mission of DAU is "to provide practitioner training, career management, and services to enable the DoD Acquisition, Technology, and Logistics (AT&L) community to make smart business decisions, and deliver timely and affordable capabilities to the warfighter."

The intent of this section is to provide an overview of the Defense Acquisition Management Framework. It will introduce the activities, phases, and efforts of the acquisition life-cycle. Cost estimate requirements will be highlighted. The section also briefly identifies the acquisition statutory and regulatory requirements. Additional policy information and references are provided at the end of this appendix. Some of the information contained in this section was obtained from the Defense Acquisition University (DAU) Acquisition 101 course and is used with permission from DAU.

[80] DoD Directive 5000.1. "The Defense Acquisition System." May 12, 2003.
[81] DoD Instruction 5000.2. "The Operation of the Defense Acquisition System." May 12, 2003.

H.1 Basic definitions

Following are some basic definitions that may be helpful in understanding the various acquisition policies and guiding documents:

- **Acquisition** – The conceptualization, initiation, design, development, test, contracting, production, deployment, logistics support, modification, and disposal of weapons and other systems, supplies, or services (including construction) to satisfy DoD needs, intended for use in or in support of military missions.
- **Systems** – The organization of hardware, software, material, facilities, personnel, data, and services needed to perform a designated function with specified results, such as the gathering of specified data, its processing, and delivery to users.
- **Risk** – A measure of the inability to achieve program objectives within defined cost and schedule constraints. Risk is associated with all aspects of the program, e.g., threat, technology, design processes, or Work Breakdown Structure elements. It has two components: the probability of failing to achieve a particular outcome, and the consequences of failing to achieve that outcome.

H.2 Acquisition authorities

Authority for the DoD to conduct systems acquisition comes from three principle sources: the law, federal acquisition regulations, and DoD acquisition policy documents. Statutory authority from Congress provides the legal basis. A list of the most prominent laws, policy documents, and other statutory information is contained at the end of this appendix.

"To retain respect for sausages and laws, one must not watch them in the making."
Otto Von Bismarck

The key reference for DoD acquisition is DoD Instruction 5000.2, Operation of the Defense Acquisition System. This instruction is consistent with statutory requirements and implements all applicable directives and guidance. It also authorizes Milestone Decision Authorities (MDAs) to tailor procedures to achieve cost, schedule, and performance goals. The MDA is the designated individual with overall responsibility and accountability for a program and has authority to approve progression through the acquisition process.

H.3 Acquisition categories

Before discussing the Acquisition Management Framework, the acquisition program categories will be explained. Acquisition Categories (ACATs), are established to determine the level of management review, decision authority, and other requirements for a program. Size, complexity, and risk generally determine the category of the program. A technology project or acquisition program is categorized based on its location in the acquisition process, dollar value, and MDA special interest. There are two separate types of ACATs: one for weapon systems and Command, Control, Communications, Computers, and Intelligence (C4I) systems; and one for Automated Information Systems (AISs). Also, some programs are not categorized as ACATs and are designated as Abbreviated Acquisition Programs (AAPs).

USD (AT&L) – Under Secretary of Defense Acquisition, Technology, and Logistics

ASD (NII) – Assistant Secretary of Defense (Networks and Information Integration)

The user who originates an Initial Capabilities Document determines if the need could potentially result in the initiation of a new program and makes a recommendation to the MDA regarding the category. The final ACAT determination is made by the appropriate MDA at the Milestone B review.

When cost growth or other factors result in reclassifying a program to a higher category, the DoD Component shall notify the USD (AT&L) or the ASD (NII) / DoD Chief Information Officer (CIO).

Designations are assigned per Department of Defense Instruction (DoDI) 5000.2 Enclosure 2 or, for the Navy, Secretary of the Navy (SECNAV) Instruction 5000.2C (SECNAVINST 5000.2C) Enclosure (2). Programs are categorized as ACAT I – IV or Abbreviated Acquisition. The review process depends upon the projected spending level. The intention is that there should be no more than two levels of review between a Program Manager and the MDA. See the tables at the end of this appendix for cost thresholds, decision authorities, and other ACAT details.

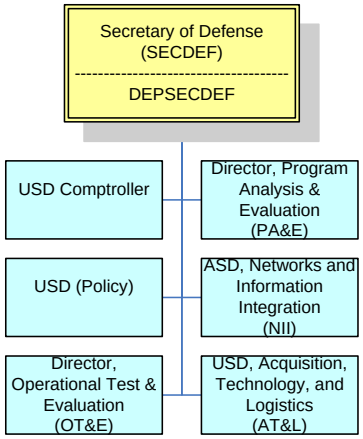


Figure H-2: Acquisition Oversight

H.3.1 ACAT I

An ACAT I program is labeled a Major Defense Acquisition Program (MDAP). A major system is a combination of elements that function together providing the required mission capabilities, including hardware, equipment, software, or any combination thereof. It does not include construction or other improvements to real property. This designation is assigned if the total expenditure for Research, Development, Test and Evaluation (RDT&E) exceeds $365 million in fiscal year (FY) 2000 constant dollars or procurement is more than $2.19 billion (FY2000 constant dollars). Note that the MDA may consider a program as one with special interest to USD (AT&L) and assign it an ACAT I designation regardless of the dollar value.

A program is generally designated as special interest because of one or more of the following factors: technology complexity; Congressional interest; a large commitment of resources; the program is critical to achieving a capability or set of capabilities; or it is a joint program.

USD (AT&L) designates weapon system and C4I MDAPs as ACAT ID or ACAT IC. USD (AT&L) is the MDA for ACAT ID programs. The "D" refers to the Defense Acquisition Board (DAB). The DoD Component ("C" is for Component) head or designated Component Acquisition Executive (CAE) is the MDA for ACAT IC programs. The CAE function includes the Service Acquisition Executives (SADs) for the military departments and acquisition executives in other DoD components. For example, the Assistant Secretary of the Navy (Research, Development, and Acquisition) or ASN (RD&A) is the Department of the Navy (DON) CAE or MDA.

"Inanimate objects are classified scientifically into three major categories – those that don't work, those that break down, and those that get lost."

Russell Baker

Major Automated Information System (MAIS) programs are designated ACAT IA by ASD (NII). An ACAT IA is divided into IAM and IAC categories. As with ACAT IC programs the "C" in IAC refers to Component. In this case, either the CAE or the Component CIO is the MDA. ASN (RD&A) is the MDA for DON ACAT IAC programs unless this authority is specifically delegated. The "M" (in ACAT IAM) refers to Major Automated Information System (MAIS).

H.3.2 ACAT II

Part of the definition for each of the successive categories is that the particular ACAT does not meet the criteria for the level above. ACAT II programs do meet the criteria for a major system and have lower cost thresholds than ACAT I (RDT&E total expenditure of more than $140

million in FY 2000 constant dollars, or for procurement , more than $660 million in FY 2000 constant dollars). ACAT II is not applicable to Automated Information System (AIS) programs. The MDA is the CAE.

H.3.3 ACAT III

ACAT III is the next lower range or cost threshold. This category includes the less-than MAIS programs. The reasons for the designation are that it does not meet the criteria for ACAT II or above and it is less-than a MAIS program. The decision authority is designated by the DoD CAE at the lowest appropriate level.

H.3.4 ACAT IV

ACAT IV programs are not mentioned in DoDI 5000.2, but are designated in SECNAVINST 5000.2C. There are two types of ACAT IV programs, IVT and IVM. "T" or Test programs require Operational Test and Evaluation (OT&E) while "M" or Monitor programs do not. The Commander Operational Test and Evaluation Force (COMOPTEVFOR) or Director, Marine Corps Operational Test and Evaluation Activity (Director, MCOTEA) may elect to monitor ACAT IVM programs. Program Executive Officers (PEOs), Systems Command (SYSCOM) Commanders, and Direct Reporting Program Managers (DRPMs) designate ACAT IV programs and may delegate MDA authority.

H.3.5 Abbreviated Acquisition Programs (AAPs)

Small DON acquisitions and modifications may be designated an AAP if they meet the cost threshold and other criteria and do not require OT&E. AAPs and IT AAPs are designed to accommodate relatively small, low-risk and low-cost acquisition and modification programs. Developing policies and procedures for review and reporting of AAPs is the responsibility of PEOs, SYSCOM Commanders, and DRPMs. APPs do not meet the criteria for ACAT IV or above.

H.4 Acquisition management framework

The DoDI 5000.2 process is commonly referred to as the Defense Acquisition Management Framework. The framework is represented by Figure H-3.

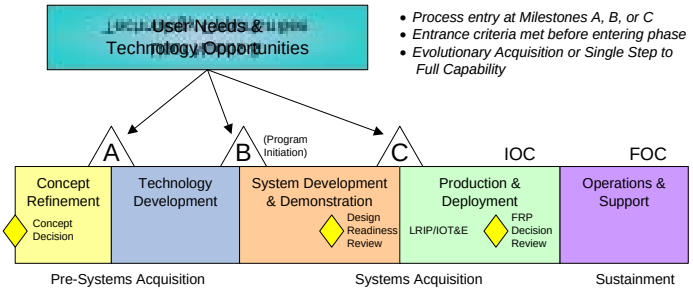


Figure H-3: Defense Acquisition Management Framework

H.4.1 Framework elements

The Acquisition Management Framework is separated into three activities: Pre-Systems Acquisition, Systems Acquisition, and Sustainment. These activities are divided into five phases: Concept Refinement, Technology Development, System Development and Demonstration, Production and Deployment, and Operations and Support. The Phases in System Acquisition and Sustainment are further divided into six efforts: System Integration, System Demonstration, Low-Rate Initial Production or limited deployment (if applicable), Full-Rate Production and Deployment, Sustainment, and Disposal. An example of a detailed framework is available at: www.dau.mil/pubs/IDA/IDA_04.aspx

The framework indicates key points in the process known as milestones. A milestone is the point at which a recommendation is made and approval sought regarding starting or continuing an acquisition program, i.e., proceeding to the next phase. The milestones established by DoDI 5000.2 are: Milestone A approves entry into the Technology Development (TD) phase; Milestone B approves entry into the System Development and Demonstration phase; and Milestone C approves entry into the Production and Deployment (P&D) phase. Also of note are: the Concept Decision (CD) that approves entry into the Concept Refinement phase; the Design Readiness Review (DRR) that ends the System Integration (SI) effort and continues the SDD phase into the System Demonstration (SD) effort; and the Full Rate Production Decision Review (FRPDR) at the end of the Low Rate Initial Production (LRIP) effort of the P&D phase that authorizes Full-Rate Production (FRP) and approves deployment of the system to the field or fleet. Note that LRIP applies to MDAPs (ACAT I) only.

The framework may be entered at multiple points. The entry points, and subsequent acquisition path, are determined by the maturity of relevant technologies and the satisfaction of specific entrance criteria established for each phase and effort. There is no single best method to accomplish the goals of the Defense Acquisition System. There is no need for all programs to follow the entire process. The flexibility and processes allow decision makers and Program Managers to customize the acquisition strategies to fit their particular program.

H.4.2 User needs and technology opportunities

The User Needs and Technology Opportunities effort is divided into two primary areas as the title implies, User Needs Activities and Technology Opportunity Activities. User Needs Activities (as shown in Figure H-4) consist of determining the desired capabilities or requirements of the system. This activity is governed by the Initial Capabilities Document, which describes gaps in capability for a particular functional or mission area. It documents the evaluation of various materiel approaches and proposes a recommended approach that best satisfies the desired capability. The capabilities must be centered on an integrated, collaborative, joint warfighting construct in accordance with the Chairman of the Joint Chiefs of Staff Instruction (CJCSI) 3170.01C, “Joint Capabilities Integration and Development System” (JCIDS).

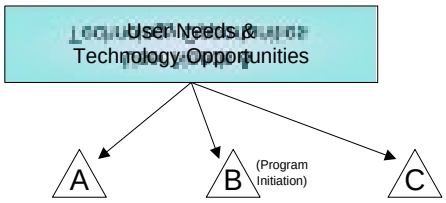


Figure H-4: User Needs Activities

The User Needs Activities also involve preparation of the Capability Description Document (CDD) and the Capability Production Document

(CPD). The CDD provides operational performance parameters necessary to design a proposed system, builds on the Initial Capabilities Document, and must be approved prior to Milestone B. The Capabilities Production Document (CPD) refines performance parameters from the CDD to support production. It must be approved prior to Milestone C by the Joint Requirements Oversight Council (JROC) or Sponsor/Domain Owner depending upon whether the program is of joint interest or not.

The other elements of the User Needs and Technology Opportunities effort are the Technology Development activities. They are conducted to ensure the transition of innovative concepts and superior technology to the user and acquisition customer. This is accomplished using Advanced Technology Demonstrations, Advanced Concept Technology Demonstrations, or Joint Warfighting Experiments.

The sponsor is the DoD component, domain owner or other organization responsible for all common documentation, periodic reporting, and funding actions required to support the capabilities development and acquisition process for a specific capability proposal.

H.4.3 Pre-systems acquisition

Pre-Systems Acquisition activities involve development of user needs, science and technology efforts, and concept refinement work specific to the development of a materiel solution to an identified, validated need. As shown in Figure H-5, the activities are governed by the Initial Capabilities Document and supported by the Concept Refinement and Technology Development phases. The acquisition strategy is developed during this activity.

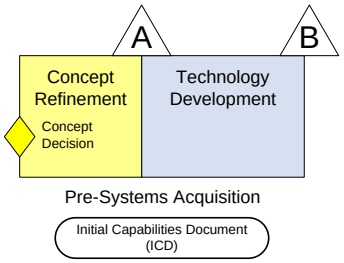


Figure H-5: Pre-Systems Acquisition

"Evolutionary acquisition is the preferred DoD strategy for rapid acquisition of mature technology for the user."[82] The acquisition strategy guides program execution from initiation through post-production support. It provides a summary description of the capability need that the acquisition of technology, products, and services are intended to satisfy. The strategy prescribes accomplishments for each acquisition phase and identifies the critical events that govern program management.

In an evolutionary approach, the user capability is divided into two or more increments, each with increased capabilities. There are two basic approaches: spiral development, and incremental development. In a spiral development, the desired capabilities are identified but the end state is unknown at program initiation. The requirements are refined through demonstration and risk management. Future increments derive requirements from user feedback and maturing technology. Incremental development has a known end state and requirements. The requirements are met over time through multiple increments dependent upon available technology.

H.4.3.1 Concept refinement phase

The acquisition process begins with refinement of the selected concept to meet a stated capability need. The intent of Concept Refinement is to put innovation into practice and foster collaboration between the warfighter, developers, testers, sustainers, and cost estimators to arrive at the best approach to solve the user's needs. This collaboration relies on Integrated Process Teams (IPTs), with representatives from the functional disciplines, to identify and resolve issues and to make recommendations.

[82] DoD Instruction 5000.2. "Operation of the Defense Acquisition System." May 12, 2003.

Concept Refinement begins with the Concept Decision. The decision to begin this phase does not mean that a new acquisition program has been initiated. Entrance into the Concept Refinement phase depends upon an approved Initial Capabilities Document and an approved plan for conducting an Analysis of Alternatives (AoA) for the selected concept. The AoA provides the basis for a Technology Development Strategy (TDS), which is submitted prior to Milestone A for potential ACAT I and IA programs.

Through studies, lab experiments, modeling, market research, and similar activities, the proposed solutions are examined. Key elements to be assessed include technology maturity and technical risk. The phase ends when the MDA approves the preferred solution resulting from the AoA and approves the associated TDS. The MDA establishes a date for Milestone A and documents all decisions in an Acquisition Decision Memorandum (ADM).

H.4.3.2 Milestone A

The Milestone A decision determines the program course. At this milestone, the MDA approves the TDS and sets criteria for the TD phase. If there is no predetermined concept and evaluation of multiple concepts is needed, the MDA will approve a Concept Exploration effort. If there is an acceptable concept without defined system architecture, the MDA will approve a Component Advanced Development effort. Milestone A approval does not initiate a new acquisition program.

H.4.3.3 Technology development phase

The purpose of this phase is to reduce technology risk and to determine the appropriate set of technologies to be integrated into the complete system. It is a continuous iterative process, requiring collaboration to determine the viability of current technical knowledge and tools while simultaneously refining user requirements. Note that shipbuilding programs may be initiated at the beginning of Technology Development in order to start ship design concurrent with sub-system / component technology development.

The Initial Capabilities Document and TDS guide the efforts during this phase. Demonstrations of the proposed technology by the developer may be necessary to show the user that the solution is affordable, militarily useful, and based upon mature technology. In an evolutionary acquisition, technology identification and development continues for future increments in parallel with acquisition of the current or preceding increments. This allows for a faster progression into the System Development and Demonstration (SDD) phase.

During the Technology Development phase, the Capability Development Document (CDD) is created. The CDD builds upon the Initial Capabilities Document and provides the necessary details to design the proposed system. The project exits the phase when an affordable increment of useful capability to the military has been identified, demonstrated in the relevant environment, and can be developed within a short timeframe (usually less than five years). The Milestone B decision follows the completion of Technology Development.

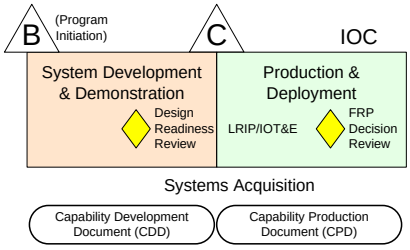


Figure H-6: Systems Acquisition

H.4.3.4 Milestone B

The purpose of Milestone B is to authorize entry into the System Development and Demonstration phase. Milestone B approval can lead to either System Integration or System Demonstration depending upon the maturity of the technology. Programs that enter the acquisition process at Milestone B must have an Initial Capabilities Document that provides the context for determination and approval of the needed capability and a CDD that describes specific program requirements.

At Milestone B, the MDA considers the following requirements: Validated CDD, System Treat Assessment, Program Protection Plan, Technology Readiness Assessment, Affordability Assessment, and Test and Evaluation Master Plan (TEMP). Also, the MDA approves: the Acquisition Program Baseline (APB), LRIP quantities (where applicable), System Development and Demonstration exit criteria, the DRR exit criteria, (if needed), and the acquisition strategy. The Acquisition Strategy requires collaboration between the Program Manager, MDA, and the functional communities engaged in and supporting DoD acquisition. It is used to guide activities during the SDD phase.

Lead ship in a class normally authorized at Milestone B

If technology maturity was demonstrated but there was no integration of the subsystems into a complete system, then the program enters the system integration effort. If the system was demonstrated by prototype articles or Engineering Development Models (EDMs), then the program enters the System Demonstration phase.

Milestone B is the point at which an acquisition program is initiated. Before initiating a new acquisition program, DoD Components must affirmatively answer the following questions:

- Does the acquisition support core/priority mission functions that need to be performed?
- Does the acquisition need to be undertaken by the DoD component because no alternative private sector or governmental source can better support the function?
- Does the acquisition support work processes that have been simplified or otherwise redesigned to reduce costs, improve effectiveness, and make maximum use of COTS technology?

The Office of the Secretary of Defense (OSD), the Military Departments, the Chairman of the Joint Chiefs of Staff, the Combatant Commands, the Office of the Inspector General of the Department of Defense, the Defense Agencies, the DoD Field Activities, and all organizational entities within the Department of Defense are collectively referred to as "the DoD Components".

H.4.4 Systems acquisition

The Systems Acquisition activity requires a great deal of planning and preparation and comprehensive knowledge of the program and the defense acquisition environment. The activity is divided into two phases: the System Development and Demonstration and Production and Deployment.

H.4.4.1 System development and demonstration

The objective of the System Development and Demonstration phase is to demonstrate an affordable, supportable, interoperable, and producible system in its intended environment. This is accomplished using EDMs or commercial items that meet validated requirements and ensure that necessary industrial capabilities to produce the systems are available. The phase can be entered directly from the TD phase as determined by the MDA. Entrance depends upon technology maturity, approved requirements, and funding. Completion of this phase is dependent upon a decision by the MDA to commit to the program at Milestone C or to end the effort.

Transition into SDD requires full funding. The SDD has two major efforts: System Integration and System Demonstration. System Integration requires a technical solution for the system. At this point, the subsystems are integrated, the detailed design is completed, and efforts are made to reduce system-level risks. The effort is guided by an approved CDD which includes a minimum set of Key Performance Parameters (KPPs). The program exits System Integration when the system has been demonstrated using prototype articles or EDMs and upon completion of a Design Readiness Review.

The Design Readiness Review form and content is determined by the MDA and provides an opportunity to assess: the design maturity, corrective actions for deficiencies, scope of testing, safety and other risks, key system characteristics, reliability, and manufacturing processes. A successful review ends the System Integration effort and moves the program into the System Demonstration effort.

System Demonstration is intended to show that the system works. The ability of the system (prototype) to operate must be shown in its intended environment. It must demonstrate that it provides the required capabilities and that it meets or exceeds the exit criteria established in the TDS.

The completion of the SDD phase is dependent on a decision by the MDA to either commit to the program at Milestone C or to end this effort.

H.4.4.2 Milestone C

The purpose of Milestone C is to authorize entry into: LRIP, production or procurement (for systems that do not require LRIP), authorizes operational testing, or limited deployment for MAIS or software-intensive systems with no production components. A partial list of Milestone C criteria includes: acceptable performance, mature software, satisfactory results from testing and the operational assessment, and no significant manufacturing risks. Other conditions include affordability, interoperability, and supportability. See DoDI 5000.2 for details.

Prior to the Milestone C decision, the MDA considers: cost and manpower estimates (for MDAPs), the system threat assessment, and environmental issues. A favorable Milestone C decision commits DoD to production of the system. At Milestone C, the MDA approves: the updated acquisition strategy, updated development APB, exit criteria for LRIP or limited deployment, and the ADM.

- MDAP – Major Defense Acquisition Program
- APB – Acquisition Program Baseline
- ADM – Acquisition Decision Memorandum

For MDAPs and major systems, Milestone C authorizes entry into LRIP; a subsequent review and decision authorizes full rate production. LRIP is not applicable for MAIS programs or software-intensive systems without developmental hardware. However, a limited deployment phase may be applicable to these systems.

H.4.4.3 Production and deployment

The purpose of the P&D phase is to achieve an operational capability that satisfies mission needs. The production requirement does not apply to MAIS programs, but software maturity must be proven prior to operational deployment. The P&D phase consists of the LRIP effort, the FRPDR, and the Full-Rate Production and Deployment effort.

The LRIP effort is intended to complete development of a manufacturing capability able to efficiently produce the necessary number of systems for Initial Operational Test and Evaluation (IOT&E). The production base must also allow for an effective increase in the production rate when operational testing is completed. The test director shall determine the number of production systems (articles) required for test and evaluation.

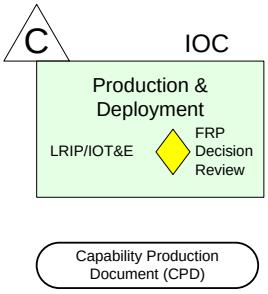


Figure H-7: Production &Deployment

The CPD guides the LRIP effort. As indicated in the Milestone B requirements, the LRIP quantities are determined by the MDA. Any increase must likewise be approved during this effort. Deficiencies found during test (prior to Milestone C) must be resolved and verified before proceeding beyond LRIP. LRIP for ships and satellites is production of items at the minimum quantity and rate that is feasible and that preserves the mobilization production base for the system.

The decision to continue to full-rate production, or limited deployment for MAIS/software-intensive systems, requires completion of IOT&E and approval of the MDA. Other reports may be necessary per DoDI 5000.2. Before the FRPDR, the MDA considers: the manufacturing process, reliability, critical processes, and other data. An Independent Cost Estimate (ICE) for MDAPs, or a Component Cost Analysis for MAISs, is required at the FRP Decision Review.

Full-Rate Production and Deployment begins after a successful FRPDR by the MDA. This effort delivers the fully funded quantity of systems, supporting materiel, and services for the program or increment to the users. The system achieves Initial Operational Capability (IOC) during this effort.

H.4.5 Sustainment

The Sustainment activity has one phase, Operations and Support (O&S), consisting of two major efforts: Sustainment and Disposal. The objective of this activity is to provide cost-effective support for the operational system over its total life-cycle. Operations and Support begins when the first systems are deployed. Since deployment begins in the latter portions of the P&D phase, these two activities overlap. Later, when a system has reached the end of its useful life, it needs to be disposed of appropriately.

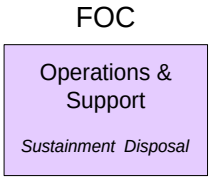


Figure H-8: Operations & Support

H.4.5.1 Sustainment effort

Thought towards maintenance and sustainability begins at system design. Wise planning during systems engineering, building for reliability and ease of maintenance all lead to effective sustainment. Sustainment includes many functions such as: supply, transportation, configuration management, training, personnel, safety, and security.

Support and life-cycle affordability plans in the acquisition strategy are implemented by the Program Managers. They address: the support and fielding requirements to meet readiness and performance objectives, lower total ownership costs, risk reduction, and avoidance of harm to health and environment. Sustainment strategies need to evolve throughout the life-cycle to take advantage of cost saving methods and improved technology. This is especially important for later increments or upgrades and modifications to the system. Decision makers must look at total ownership costs within the context of overall DoD priorities.

> *The cost of maintaining systems in many organizations has been observed to range from 40% to 70% of resources allocated to the entire software life-cycle.*
>
> *Penny Grubb and Armstrong Takang, from* Software Maintenance: Concepts and Practice

H.4.5.2 Disposal effort

At the end of the system's useful life, it must be demilitarized and disposed of in accordance with all the requirements and policies relating to safety, security, and the environment. These plans are also addressed by the Program Managers in the acquisition strategy. Sustainment continues until all elements of the system are removed from the inventory.

H.5 Cost analysis

The Defense Acquisition System documents refer to life-cycle cost, total ownership cost, and various cost estimates and analyses. For a defense acquisition program, life-cycle cost consists of research and development, investment (i.e., LRIP and P&D phases), O&S, and disposal costs over the entire life of the system. The life-cycle cost includes the direct acquisition costs and any indirect costs that can be logically attributed to the program. Total ownership cost includes the elements of life-cycle cost along with the infrastructure or business process costs not necessarily attributable to the program.

Cost estimates and analyses are directed by law and governed by DoD and service directives and instructions. Title 10 of the United States Code, Subtitle A, Part IV, Chapter 144, Section 2434, states:

> *The Secretary of Defense may not approve the system development and demonstration, or the production and deployment, of a major defense acquisition program unless an independent estimate of the full life-cycle cost of the program and a manpower estimate for the program have been considered by the Secretary.*

In DoD Directive 5000.4, *Cost Analysis Improvement Group* (CAIG), the specific responsibility for fulfilling the requirement for an independent cost estimate is assigned to the Office of the Secretary of Defense (OSD) Cost Analysis Improvement Group for ACAT ID programs, pre-MDAP projects approaching formal program initiation as a likely ACAT ID program, and ACAT IC programs when requested by the USD (AT&L).

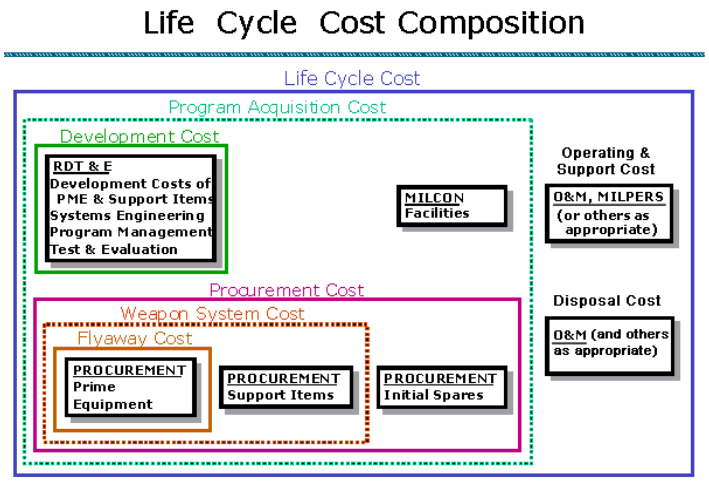


Figure H-9: Life Cycle Cost Composition

DoDI 5000.2 specifies that the CAIG independent cost estimate will be provided in support of major milestone decision points (Milestone B, C, or the FRPDR). In addition, the MDA may request, at any time, that the CAIG prepare other ICEs or conduct other ad-hoc cost assessments for programs subject to DAB review or oversight. The OSD CAIG reviews cost estimates prepared by the program office and/or the DoD Component cost agency. Overall, the CAIG serves as the principal advisory body to the MDA on all matters concerning an acquisition program's life-cycle cost.

DoDI 5000.2 Enclosure 3 lists cost information required for the milestones and reviews. For example, an ICE is required for MDAPs (n/a for AISs) at Milestone B, Milestone C, and the FRPDR. The instruction also requires that a program office estimate (POE) and a DoD Component cost analysis

estimate be prepared in support of acquisition milestone reviews. Of note, only a cost assessment is required at program initiation for ships. The table in the enclosure also points out the Affordability Assessment, Economic Analysis, Component Cost Analysis, and what type of programs they are required for and at what points in the program.

As stated, the OSD CAIG prepares independent life-cycle cost estimates for major defense acquisition programs at major milestone reviews. It also concurrently reviews cost estimates prepared by the program office and/or the DoD Component cost agency. The CAIG has other responsibilities, as described in DoDD 5000.4. One of the major responsibilities is to establish guidance on the preparation of life-cycle cost estimates subject to CAIG review. This guidance includes standard definitions of cost terms in the management of DoD acquisition programs.

H.5.1 Cost estimating

Cost estimating and analysis are clearly essential activities in determining life-cycle and total ownership costs. The estimates must include all costs associated with the system or the total "Life Cycle Cost Composition" (Figure H-9). It is, therefore, important to submit well-documented cost estimates that are ready for review. In general, the documentation should be sufficiently complete and well organized so that a cost professional can replicate the estimate, given the documentation.

Cost estimates are based on the program definition. For ACAT I and IA programs, the Cost Analysis Requirements Description (CARD) is used to formally describe the acquisition program (and the system itself) for purposes of preparing the program office cost estimate, Component cost position (if applicable), the ICE, and whenever an Economic Analysis is required. The Cost Analysis Guidance and Procedures (DoD 5000.4-M) says the CARD should be considered a "living" document that is updated to reflect changes.

> *Requirements are like water. They're easier to build on when they're frozen.*
>
> Anonymous

As an aside, requirements uncertainty, completeness, and estimating techniques are discussed elsewhere in this guidebook. Suffice it to say, "Stable requirements are the holy grail of software development."[83] The level of detail presented in the CARD will vary depending upon the maturity of the program and will affect the fidelity of the cost estimate. During the development process, customers and developers gain a better understanding of the system and their needs, which leads to requirements change. Consequently, techniques used to develop the cost estimates need to take into account the stage of the acquisition cycle that the program is in when the estimate is made. During the early phases of the program, it is expected that the use of parametric (statistical) costing techniques will be used for the development of the cost estimates.

> Arthur C. Clarke's 3 Laws:
>
> 1. When a distinguished but elderly scientist states that something is possible, he is certainly right. When he states that something is impossible, he is very probably wrong.
> 2. The only way of discovering the limits of the possible is to venture a little way past them into the impossible.
> 3. Any sufficiently advanced technology is indistinguishable from magic.
>
> Revised Edition of *Profiles of the Future* (1973)

Cost estimates must capture all costs of the program, regardless of fund source or management control; they are not to be arbitrarily limited to certain budget accounts or to categories controlled by certain lines of authority.

When programs are less mature (in Concept Refinement, Technology Development, or System Development & Demonstration), program cost

[83] McConnell, Steve. Code Complete, Second Edition. Redmond, WA: Microsoft Press, 2004.

estimates that are supporting the acquisition system normally are focused on life-cycle cost or elements of life-cycle cost. Examples include: affordability assessments, analyses of alternatives, cost-performance tradeoffs, and program cost goals. More refined life-cycle cost estimates are often used within the program office to support internal decision-making such as design change evaluations and assessments of production, reliability, maintenance, and support. As programs mature (transition from Production and Deployment to Operations and Support), cost estimates will likely be expanded in scope to encompass total ownership costs.

For cost elements which are determined to be low-risk and low-cost based on an independent analysis of the program assumptions, the CAIG Chair may authorize the use of the Sufficiency Review method for assessing the adequacy of these cost elements. The review includes an evaluation of the techniques and data used to develop the POE and, if available, the use of data from alternative sources to verify the POE. The results of the review are documented and provided to the CAIG.

H.5.2 Estimate types

DoD, service component, and other acquisition guidance provides more detailed information on the requirements for the various estimates that are part of the Defense Acquisition System process. For the purposes of this guidebook, a brief summary of the types mentioned in DoDI 5000.2, DoD Directive 5000.4, and DoD 5000.4-M are included here. However, these estimate type definitions are common to most other documentation.

H.5.2.1 Life-cycle cost estimate

DoDD 5000.1 directs estimation of ownership costs to begin as early as possible in the acquisition process. The Life-Cycle Cost Estimate (LCCE) must be provided to the MDA before approval for an MDAP to proceed with the Systems Acquisition activity. Life-Cycle Cost includes all program elements; all affected appropriations; and encompasses the costs, contractor and in house effort, as well as existing assets to be used, for all cost categories. It is the total cost to the government for a program over its full life, and includes the cost of research and development, investment in mission and support equipment (hardware and software), initial inventories, training, data, facilities, etc., and the operating, support, and, where applicable, demilitarization, detoxification, or long term waste storage. The Independent Cost Estimate and the Program Office Estimate described below are both, essentially, LCCEs. The difference between them is the organization who prepares it.

H.5.2.2 Total Ownership Cost

As explained earlier, TOC consists of the life-cycle cost, as well as other infrastructure or costs not necessarily attributable to the program. Infrastructure is interpreted in the broadest sense to include all military and defense agency activities which sustain the military forces assigned to the combatant and component commanders. The major categories of infrastructure are: equipment support (acquisition and central logistics activities), military personnel support (non-unit central training, personnel administration and benefits, and medical care), and military base support (installations and communications/information infrastructure). Other costs include many things such as environmental and safety compliance and contract oversight.

The CDD and CPD include a program affordability determination identified as life-cycle cost or, if available, total ownership cost. The APB should contain cost parameters (objectives and thresholds) for major elements of program life-cycle costs (or TOCs, if available).

In general, traditional life-cycle cost estimates are in most cases adequate in scope to support decisions involving system design characteristics (such as system weight, material mix, or reliability and maintainability). However, in special cases, depending on the issue at hand, the broader perspective of total ownership cost may be more appropriate than the life-cycle cost perspective, which may be too narrow to deal with the particular context.

H.5.2.3 Analysis of Alternatives

The AoA is the evaluation of the operational effectiveness, operational suitability, and estimated cost of alternative systems to meet a mission capability. The AoA is developed during the Concept Refinement phase and is required for Milestone A. For potential and designated ACAT I and IA programs, the Director, Program Analysis and Evaluation will direct development of the AoA by preparing initial guidance and reviewing the analysis plan and final analysis products.

H.5.2.4 Independent Cost Estimate

An ICE is required for major milestone decision points. Estimates, independent of the developer and the user, ensure an impartial cost evaluation. Acquisition officials typically consider these assessments when making decisions.

Gregory Benford's Corollary to Clarke's Third Law: Any technology distinguishable from magic is insufficiently advanced.

An ICE is prepared and provided to the MDA before an MDAP proceeds with System Acquisition activities. The OSD CAIG usually prepares the ICE. However, the component cost agency, for example the Naval Center for Cost Analysis (NCCA), is responsible for the ICE when it is not prepared by the CAIG.[84] Similarly, in the case of ACAT II programs, an ICE is prepared by the SYSCOM/PEO Cost Estimating Office.

H.5.2.5 Program Office Estimate (POE)

The program office prepares many estimates, as can be derived from the previous information, such as the AoA and affordability assessment. They prepare an LCCE for all ACAT I program initiation decisions and at all subsequent program decision points. This is known as the Program Office Estimate (POE). The CAIG may incorporate in its estimate, with or without adjustment, specific portions of the POE or the DoD Component Cost Analysis estimate, if it has independently established that the portions included are valid.

H.5.2.6 Component Cost Analysis

Another type of independent estimate is called the Component Cost Analysis. It is mandatory for MAIS programs and as requested by the CAE for MDAP programs. The need for a Component Cost Analysis at Milestone A is evaluated for each program in tailoring the oversight process. The Component Cost Analysis is normally prepared by the Service Component

[84] SECNAVINST 5000.2C. Implementation and Operation of the Defense Acquisition System and the Joint Capabilities Integration and Development System. Nov. 19, 2004.

Cost Estimating Agency or an entity not associated with the program office or its immediate chain of command. Note that recent references (DoD Instruction [DoDI] 5000.2 and the Defense Acquisition Guidebook) define CCA as the Clinger-Cohen Act and spell out Component Cost Analysis.

H.5.2.7 Economic Analysis

The purpose of the Economic Analysis is to determine the best acquisition alternative. This is done by assessing the net costs and benefits of the proposed program relative to the status quo. An Economic Analysis is only required for MAIS programs. Anytime one is required, the DoD Component responsible for the program may also be required to provide a DoD Component Cost Analysis. Normally, the Economic Analysis is prepared by the AIS program office. The CARD is used to define and describe the AIS program for purposes of preparing both the Economic Analysis and the DoD Component Cost Analysis.

H.6 Acquisition category information

The following tables were copied from the NAVAIR Training Systems Division website. The website table data was last updated 10 January 2005.

Table H-1: DoD Instruction 5000.2 ACATs

Description and Decision Authority for ACAT I - III Programs		
Acquisition Category	**Reason for ACAT Designation**	**Decision Authority**
ACAT I	▪ MDAP (10 USC 2430, reference (n)) • Dollar value: estimated by the USD(AT&L) to require an eventual total expenditure for RDT&E of more than $365 million in fiscal year (FY) 2000 constant dollars or, for procurement, of more than $2.190 billion in FY 2000 constant dollars • MDA designation ▪ MDA designation as special interest	ACAT ID: USD(AT&L) ACAT IC: Head of the DoD Component or, if delegated, the DoD Component Acquisition Executive (CAE)
ACAT IA	▪ MAIS: Dollar value of AIS estimated by the DoD Component Head to require program costs (all appropriations) in any single year in excess of $32 million in fiscal year (FY) 2000 constant dollars, total program costs in excess of $126 million in FY 2000 constant dollars, or total life-cycle costs in excess of $378 million in FY 2000 constant dollars ▪ MDA designation as special interest	ACAT IAM: ASD(C3I)/DoD CIO ACAT IAC: CAE, as delegated by the DoD CIO
ACAT II	▪ Does not meet criteria for ACAT I ▪ Major system • Dollar value: estimated by the DoD Component Head to require an eventual total expenditure for RDT&E of more than $140 million in FY 2000 constant dollars, or for procurement of more than $660 million in FY 2000 constant dollars (10 USC 2302d, reference (o)) • MDA designation4 (10 USC 2302(5), reference (p)) ▪ MDA designation as special interest	DoD CAE or the individual designated by the CAE
ACAT III	▪ Does not meet criteria for ACAT II or above ▪ Less-than a MAIS program	Designated by the DoD CAE at the lowest level appropriate

Notes:
1. In some cases, an ACAT IA program, as defined above, also meets the definition of an MDAP. The USD(AT&L) and the ASD(C3I)/DoD CIO shall decide who will be the MDA for such programs. Regardless of who is the MDA, the statutory requirements that apply to MDAPs shall apply to such programs.
2. An AIS program is an acquisition program that acquires IT, except IT that involves equipment that is an integral part of a weapon or weapons system, or is an acquisition of services program.
3. The ASD(C3I)/DoD CIO shall designate programs as ACAT IAM or ACAT IAC. MAIS programs shall not be designated as ACAT II.
4. As delegated by the Secretary of Defense or Secretary of the Military Department.

Table H-2: SECNAV Inst 5000.2C ACATs

Description and Decision Authority for ACAT I-IV and AAP Programs		
Acquisition Category	**Criteria for ACAT or AAP Designation**	**Decision Authority**
ACAT I	▪ Major Defense Acquisition Programs (MDAPs) (10 USC 2430) • RDT&E total expenditure > $365 million in FY 2000 constant dollars, or • Procurement total expenditure > $2.190 billion in FY 2000 constant dollars, or ▪ USD(AT&L) designation as special interest	ACAT ID: USD(AT&L)ACAT IC: SECNAV, or if delegated, ASN(RD&A) as the CAE
ACAT IA	▪ Major Automated Information Systems (MAISs) • Program costs/year (all appropriations) > $32 million in FY 2000 constant dollars, or • Total program costs > $126 million in FY 2000 const. dollars, or • Total life-cycle costs > $378 million in FY 2000 constant dollars ▪ ASD(NII) designation as special interest	ACAT IAM: ASD(NII)/DoD CIOACAT IAC: ASN(RD&A), as delegated by the DoD CIO
ACAT II	▪ Does not meet the criteria for ACAT I ▪ Major Systems (10 USC 2302(5)) • RDT&E total expenditure > $140 million in FY 2000 constant dollars, or • Procurement total expenditure > $660 million in FY 2000 constant dollars, or ▪ ASN(RD&A) designation as special interest ▪ Not applicable to IT system programs	ASN(RD&A), or the individual designated by ASN(RD&A)
ACAT III	▪ Does not meet the criteria for ACAT II or above ▪ Weapon system programs: • RDT&E total expenditure = $140 million in FY 2000 constant dollars, or • Procurement total expenditure = $660 million in FY 2000 constant dollars, and • Affects mission characteristics of ships or aircraft or combat capability ▪ IT system programs: • Program costs/year = $15 million = $32 million in FY 2000 constant dollars, or • Total program costs = $30 million = $126 million in FY 2000 constant dollars, or • Total life-cycle costs = $378 million in FY 2000 constant dollars	Cognizant PEO, SYSCOM Commander, DRPM, or designated flag officer or senior executive service (SES) official. ASN(RD&A), or designee, for programs not assigned to a PEO, SYSCOM, or DRPM.
ACAT IVT	▪ Does not meet the criteria for ACAT III or above ▪ Requires operational test and evaluation ▪ Weapon system programs: • RDT&E total expenditure = $140 million in FY 2000 constant dollars, or • Procurement total expenditure = $660 million in FY 2000 constant dollars ▪ IT system programs: • Program costs/year < $15 million, or • Total program costs < $30 million, or • Total life-cycle costs = $378 million in FY 2000 constant dollars	Cognizant PEO, SYSCOM Commander, DRPM, or designated flag officer, SES official, or Program Manager. ASN(RD&A), or designee, for programs not assigned to a PEO, SYSCOM, or DRPM
ACAT IVM	▪ Does not meet the criteria for ACAT III or above ▪ Does not require operational test and evaluation as concurred with by the Office of Technology Assessment (OTA) ▪ Weapon system programs: • RDT&E total expenditure = $10 million = $140 million in FY 2000 constant dollars, or • Procurement expenditure = $25 million/year = $50 million total = $660 million total in FY 2000 constant dollars ▪ Not applicable to IT system programs	Cognizant PEO, SYSCOM Commander, DRPM, or designated flag officer, SES official, or Program Manager. ASN(RD&A), or designee, for programs not assigned to a PEO, SYSCOM, or DRPM
Abbreviated	▪ Does not meet the criteria for ACAT IV or above	Cognizant PEO, SYSCOM

Description and Decision Authority for ACAT I-IV and AAP Programs		
Acquisition Category	**Criteria for ACAT or AAP Designation**	**Decision Authority**
Acquisition Program	▪ Does not require operational test and evaluation as concurred with in writing by OTA ▪ Weapon system programs: • Development total expenditure < $10 million, and • Production or services expenditure < $25 million/year, < $50 million total ▪ IT system programs: • Program costs/year < $15 million, and • Total program costs < $30 million	Commander, DRPM, or designated flag officer, SES official, or Program Manager. ASN(RD&A), or designee, for programs not assigned to a PEO, SYSCOM, or DRPM.

H.7 Acquisition references

The following information provides additional details and resources on acquisition related topics.

H7.1 Online resources

Defense Acquisition Guidebook (DAG): The Defense Acquisition Guidebook is designed to complement the policy documents by providing the acquisition workforce with discretionary best practices that should be tailored to the needs of each program.
Acquisition professionals should use the Guidebook as a reference source supporting their management responsibilities. Link: http://akss.dau.mil/dag/DoD5000.asp?view=document&doc=3

Introduction to Defense Acquisition Management (ACQ 101): This course provides a broad overview of the DoD systems acquisition process, covering all phases of acquisition. It introduces the JCIDS and resource allocation processes, the DoD 5000 Series documents governing the defense acquisition process, and current issues in systems acquisition management. Link (to this and other DAU course descriptions): http://www.dau.mil/schedules/schedule.asp

Defense Acquisition Policy Center: References and links to Acquisition Policies (DoD, Joint Chiefs, and Services), Guidance, and Support (Tutorials, briefings, etc.). Link: http://akss.dau.mil/dapc/index.html

Acquisition Community Connection (ACC): Other resources for acquisition information, knowledge and information sharing forums to connect with others in your field, collaborative and private workspaces. Link: https://acc.dau.mil/simplify/ev_en.php

Acquisition, Technology, and Logistics Knowledge Sharing System (AKSS): Contains links to policies and tools and other suggested reading resources. Includes links to new policy documents and the latest news for DoD and the services. Link: http://akss.dau.mil/jsp/default.jsp

H7.2 Statutory information

Statutory authority from Congress provides the legal basis for systems acquisition. Some of the most prominent laws are:

- Armed Services Procurement Act (1947), as amended

- Small Business Act (1963), as amended
- Office of Federal Procurement Policy Act (1983), as amended
- Competition in Contracting Act (1984)
- DoD Procurement Reform Act (1985)
- DoD Reorganization Act of 1986 (Goldwater-Nichols)
- Federal Acquisition Streamlining Act (FASA) of 1994
- Clinger-Cohen Act of 1996

Armed Services Procurement Act - The modern era of congressional involvement in acquisition began with the Armed Services Procurement Act of 1947. The purpose of this law was to standardize contracting methods used by all of the services. As a result, the first joint DoD regulation was created—the Armed Services Procurement Regulation (ASPR). This Act continued the sealed bid as the preferred method of procurement, placed procurement rules in one location and gave us the ASPR, which was the beginnings of today's rulebook, the Federal Acquisition Regulation (FAR).

Office of Federal Procurement Policy Act of 1983 - Established a central office to define overall government contracting and acquisition policy and to oversee the system, among other things.

Competition in Contracting Act of 1984 - Revised government policy to mandate competition and created an advocate for competition, the Competition Advocate General. While competition has always been the hallmark of the system, it was not until the passage of the Competition in Contracting Act (CICA) of 1984, which mandated full and open competition, that over 50 percent of the dollars spent were actually competed. CICA instituted a very structured process for sole source authorization. It requires approval by the local competition advocate for lower dollar acquisitions.

DoD Procurement Reform Act 1985 - Defense Procurement Reform Act established a uniform policy for technical data and created a method for resolving disputes.

DoD Reorganization act of 1986 (commonly referred to as Goldwater-Nichols Act) - Among other items, revised the Joint Chiefs of Staff role in acquisition and requirements determination.

Federal Acquisition Streamlining Act - Revolutionary in its impact on the federal acquisition process. It repealed or substantially modified more than 225 statutes and pushed the contracting process into the 21st century. Among other things, it simplified the federal procurement process, reduced paperwork burdens, and transformed the simplified acquisition process to electronic commerce. Military specifications and standards are no longer the preferred method of doing business. Congress, at the DoD's urging, passed this legislation to remove some of the barriers.

Cohen-Clinger Act of 1996 - It provides that the government information technology shop be operated exactly as an efficient and profitable business would be operated. Acquisition, planning and management of technology must be treated as a "capital investment." While the law is complex, all consumers of hardware and software in the DoD should be aware of the CIO's leadership in implementing this statute. CCA emphasizes an integrated framework of technology aimed at efficiently performing the business of the DoD. Just as few businesses can turn a profit by allowing

their employees to purchase anything they want to do any project they want, the DoD also cannot operate efficiently with hardware and software systems purchased on an "impulse purchase" basis and installed without an overall plan. All facets of capital planning are taken into consideration just as they would be in private industry:

- cost/benefit ratio
- expected life of the technology
- flexibility and possibilities for multiple uses

The act included changes to competition practices, commercial item acquisition, and fundamental changes in how information technology equipment is purchased. Note: before FASA could be fully implemented, this Act became law and corrected some deficiencies in the earlier legislation and made more changes.

Defense Procurement Improvement Act of 1986 - Provided policy on the costs contractors submitted to the government for payment and on conflicts of interest involving former DoD officials.

Defense Acquisition Improvement Act of 1986 - Among other things, created the Under Secretary of Defense (Acquisition and Technology).

Ethics Reform Act of 1989 - As a result of the "ill-wind" procurement scandal, Congress mandated more stringent ethics laws.

Defense Acquisition Workforce Improvement Act of 1990 - Mandated education, training and professional requirements for the defense acquisition corp.

Federal Acquisition Reform Act of 1996 - Revised procurement laws facilitate more efficient competition; included improving debriefings, limiting need for cost/pricing data and emphasizing price versus cost negotiations, among other items.

Acquisition policies, procedures, and operations within DoD are governed by two documents. The first regulation is DoDD 5000.1, *The Defense Acquisition System*. The second is DoDI 5000.2, *Operation of the Defense Acquisition System.*

DoD Directive 5000.1*, The Defense Acquisition System* – Identifies the key officials and panels for managing the system and provides broad policies and principles that guide all defense acquisition programs.

DoD Instruction 5000.2*, Operation of the Defense Acquisition System* – Establishes a simplified and flexible management framework for translating joint capability needs and technological opportunities into stable, affordable, and well-managed acquisition programs. It applies to all defense technology projects and acquisition programs, although some requirements where stated apply only to MDAPs and MAISs.
It implements the policies and principles set forth in the directive.

There are other laws, regulations, and guidance that apply to acquisition, such as the FAR (which has a DoD supplement) and laws relating to Environment, Safety, and Occupational Health (ESOH). CJCSI 3170.01E deals with the JCIDS, which is a process to determine the capabilities needed

by the warfighter, and, ultimately, the requirements of the systems we acquire.

Defense Acquisition Guidebook - provides non-mandatory guidance on best practices, lessons learned, and expectations.

Federal Acquisition Regulation - the primary regulation for use by all Federal agencies for the acquisition of supplies and services with appropriated funds. The FAR guides and directs DOD Program Managers in many ways including acquisition planning, competition requirements, contract award procedures, and warranties. DOD has supplemented the FAR to describe its own procedures with the **Defense Federal Acquisition Regulation Supplement (**DFARS).

Environment, Safety, and Occupational Health Requirements (ESOH) - The Program Manager must ensure the system can be produced (Service specific issue), tested, fielded, operated, trained with, maintained, and disposed of in compliance with environment, safety, and occupational health (ESOH) laws, regulations and policy (collectively termed ESOH requirements). The goal of ESOH law is to protect human health and safeguard the environment. The DoD goal is to safeguard the environment, reduce accidents, and protect human health. The most effective means to meet this DoD goal is by managing risks.
PMs should also be aware how different aspects of laws, regulations and policies affect their ability to implement this goal over the system's life-cycle. Acquisition program offices need to address ESOH requirements because they can impact the cost, schedule, and performance of the system. As part of risk reduction, the Program Manager is responsible for identifying the applicable ESOH requirements and ensuring that they consult with the user to make informed decisions about whether, or to what degree, the system conforms to the applicable ESOH requirements.

Effective ESOH Risk Management Practices - DoD acquisition policy requires the Program Manager to document the program process, the schedule for completing National Environmental Policy Act (NEPA) documentation, and the status of ESOH risk management in a Programmatic ESOH Evaluation (PESHE). The PESHE is not intended to supersede or replace other ESOH plans, analyses, and reports (e.g., System Safety Management Plans/Assessments, Hazardous Materials Management Plans, NEPA documents, Health Hazard Assessments [HHA], etc.); it is a management and reporting tool for the Program Manager.

ESOH Requirements - ESOH requirements are driven from the top by federal (including Executive Orders), state, and local ESOH laws and implementing regulations. DoD and Component policies implement the law, and frequently include additional requirements. ESOH requirements:

- Establish the methods and mechanisms of a process for compliance.
- May impose civil injunctions resulting in program delays if the compliance process is not properly planned and executed.
- Mandate legal compliance requirements and processes.
- Assign compliance responsibility, primarily to the facility and installation managers and maintainers.
- For acquisition programs the Program Manager is responsible for considering ESOH requirements and their effects throughout the system life-cycle when making design decisions.

- Impose criminal and/or civil penalties for lack of compliance, which may or may not be applicable to federal agencies and individuals.

System design and functionality, materials used, and operational parameters drive the extent of ESOH compliance requirements. ESOH impacts must be considered as an integral part of any acquisition systems engineering effort. Incorporating safety and human systems engineering into design enhances performance and lowers the risk of mishaps.

National Environmental Policy Act (NEPA) - The major law directly affecting DOD systems acquisition management is the NEPA, requiring the DOD to:

- Provide full disclosure of possible impacts, alternatives, and mitigation measures.
- Consider the environment in its decisions.
- Inform and involve the public in that process.
- Seek less environmentally damaging ways to accomplish the mission or operation.
- Support informed decisions with quality documents. All acquisition programs, regardless of size, must evaluate whether the development, testing, production, fielding, operation and maintenance, and disposal of the system will affect the environment.

PMs should design the system in such a way as to minimize any negative impact on human health and the environment. DoD policy requires the Program Manager of each systems acquisition program to prepare a NEPA Compliance Schedule. The NEPA Compliance Schedule is required as part of the PESHE, and should be integrated into the overall program schedule.

H7.3 Acquisition decision support systems

Planning, Programming, Budgeting, and Execution - The Planning, Programming, Budgeting and Execution (PPBE) process is DOD's primary resource allocation process that:

- Is a calendar-driven process used for securing funding for a major acquisition program.
- Offers the basis for informed affordability assessment and resource allocation decisions.
- Provides a formal, systematic structure for making decisions on policy, strategy, and the development of forces and capabilities to accomplish anticipated missions.

Joint Capabilities Integration and Development System - The JCIDS:

- Is driven by warfighting deficiencies or needs.
- Determines mission requirements and strategies for meeting those requirements.
- Provides the basis for establishing priorities.

Acquisition Management System - The Acquisition Management System:

- Establishes a management process to translate user needs and technological opportunities into reliable and sustainable systems that provide capability to the user.
- Is an event-driven process that emphasizes risk management.
- Involves the process of periodic review and approval of programs to progress into subsequent phases of the acquisition life cycle.
- Provides a streamlined management structure.
- Links milestone decisions to demonstrated accomplishments.

Real knowledge is to know the extent of one's ignorance.

Confucius

Appendix I

Data Collection

Historical software data from the development process is crucial to cost analysts in predicting and validating size, cost, and schedule for existing and future software development projects. The data collection approach described here is intended to populate a repository for data collected during the software development process.

The data contains key information for each individual software component or Computer Software Configuration Item (CSCI) within a single project. Data is collected at various maturity levels during the development process. Ideally, data should be collected at each of the major development milestones, starting with the concept phase through final qualification testing, to assist in the resolution of the project status and growth. A fundamental principle behind any data collection effort is validation of all data prior to it becoming part of the database. Without validation, the data is meaningless and of no value. The process involves establishing a common set of references (definitions and timeframes) for the possible estimating models, determining the development environment, and carefully examining the data to ensure it is factual and realistic.

The data collected for each CSCI describes the basic information such as product sizing (new, modified, and reused code), the development effort and maturity, and the development environment (language, experience, and so forth). The data fields should allow easy input regardless of the model used to create the data.

I.1 Software data collection overview

The purpose of this series of data collection definitions and instructions is to obtain the characteristics of software projects and their respective development environments for future use in software cost estimating. This data can be included in a database of software product sizes, schedules, effort, environment, and other factors that analysts can use to compare and validate software cost estimates for new and developing systems.

I.1.1 Model comparisons

The set of estimating models used in the data collection definitions include the PRICE S™ model, the Galorath Incorporated System Evaluation and Estimation of Resources - Software Estimating Model™ (SEER-SEM™) model (now SEER™ for Software), the COnstructive COst MOdel (COCOMO/COCOMO II) developed by Dr. Barry Boehm et al, the Software Engineering Inc. Sage™ model developed by Dr. Randall Jensen, the REVised Intermediate COCOMO (REVIC) model developed by Raymond Kile and the Air Force Cost Analysis Agency (AFCAA), and the Quantitative Software Management Software Lifecycle Management Model (SLIM®) developed by Larry Putnam. Instructions and definitions within this section include a table that provides a cross-reference between the models to help clarify the data entries and relationships.

Most common estimating models use similar parameters to describe project development environments. The set of parameter descriptions used in this data collection section focuses on six widely used models as the context for describing CSCIs. However, definitions and rating scales for the individual parameters vary between models. For example, the Analyst Capability (ACAP) parameter definition for a value of "0" in the data collection model corresponds to a Sage value of "Highly motivated AND experienced team organization" and a SEER-SEM™ value of "Near Perfect Functioning Team (90th Percentile)." The Sage model range for ACAP is from 0.71 to 1.46. The SEER-SEM™ range is from Very Low to Very High. The database normalizes the model scales by using a value from 0 to 10 for all estimating environment parameters.

Table I-1: Estimating model parameter comparison

Tool	Personnel								Support							
	ACAP	PCAP	PROFAC	AEXP	DEXP	LEXP	PEXP	TEXP	DSYS	DVOL	MODP	PIMP	PVOL	RUSE	SCED	TOOL
COCOMO II	X	X		APEX	VEXP	LTEX	X			PVOL		X	X	X	X	X
PRICE-S	CPLX1	ACAP	X	CPLX1		CPLX1										CPLX1
REVIC	X	X		X	VEXP	X	PLEX/ VEXP			VIRT	X		VIRT	X	X	X
Sage	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X
SEER-SEM	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X
SLIM			P1													

Tool	Management					Product													
	MCLS	MORG	MULT	PCON	RLOC	CPLX	DATA	DISP	HOST	INTEGE	INTEGI	MEMC	PLAT	RELY	RTIM	RVOL	SECR	TIMC	TVOL
COCOMO II			SITE	X		X	X					STOR		X				TIME	
PRICE-S			CPLXM			X				X	X	UTIL	PLTFM			X		UTIL	
REVIC	SECU		SITE			X	X					STOR	X	X		X		TIME	
Sage	X	X	X		X	X		X	X			X		X	X	X	X	X	X
SEER-SEM	X	X	X		X	X		X	X			X		X	X	X	X	X	X
SLIM																			

None of the estimating models have identical parameter sets, for example, the personnel environment. The Sage, SEER-SEM™ and REVIC models use a Language Experience (LEXP) parameter, the PRICE S™ model uses a CPLX1 parameter, and COCOMO II uses a Language and Tools Experience (LTEX) parameter. Occasionally, estimating model parameters from one

model do not map directly to another or a parameter may include the effects of two or more parameters. For example, the REVIC VEXP (Virtual system experience) parameter collectively represents the SEER-SEM™ DEXP (Development system experience) and PEXP (Practices experience) parameters. Table I-1 compares the major estimating model parameters used in the data collection discussion.

I.1.2 Format

The data collection form description defines the full set of data elements necessary to describe a software development project. The data elements are compatible with commonly used cost estimating models, enabling easy transfer of data using familiar terms and definitions. The data for each CSCI includes: (1) CSCI description, (2) size data, (3) development environment data, (4) software development actuals – cost and schedule, and (5) data validation results.

The bulk of the data collection form is used to collect size data for the software development project and the environment in which the software product is developed.

Development effort, as implemented in the major estimating models, is in the form:

$$E_d = C_l C_k S_e^{\beta} \qquad \text{(I-1)}$$

where the development effort (E_d), measured from the start of full-scale development (Software Requirements Review [SRR]) through successful completion of acceptance testing by the developer, is the product of the development environment impact and the adjusted effective size. The terms are defined as:

C_1 = scaling constant,

C_k = environment adjustment factor,

S_e = effective software size, and

β = entropy factor (usually about 1.2) representing a size penalty for large software developments.

It is necessary to collect both size data and environment factors to accurately relate the development effort and productivity to the software product.

The effective software development size is a measure of the work required to produce a software product based on an equivalent product containing only new source lines of code or function points.

The development environment adjustment factors account for a multitude of impacts on the development effort due to developer capability, languages, and tools, as well as constraints imposed on the developer such as product requirements, development standards, and pre-existing product limitations. There are a total of 35 environment adjustment factors as depicted in Table I-1 that can be collected in the data collection form. The form allows the user to specify the factors to be collected according to the dictates of one of six formats: (1) COCOMO/ COCOMO II, (2) Price-S, (3) REVIC, (4) Sage, (5) SEER-SEM™ and (6) SLIM®. The selected format is specified by the user as a data element in the collection form.

The CSCI level is the level defined by independent requirements specifications (SRS) and interface control specifications (IRS). The CSCI

component level is compatible with estimating models and is the project level for tracking and comparison of software functionality represented by an independent task.

I.1.3 Structure

The data collection structure is arranged into three primary data groups: project summary data, CSCI information, and environment information. The specific CSCI data contains the general "CSCI Information" (a row in the spreadsheet representation of Table I-2) and the "Environment" information.

Table I-2: CSCI information

PROJ	CSCI ID	SUB CSCI ID	VER	Product CSCI Attributes				
				Contractor	POC			Description
					Name	Phone	E-mail	
Milstar	Bus	Antenna C+1	7	ACME	J.P. Jones	310-555-1324	jpjones@acme.com	2014 virtual antenna control

Project Summary Data identifies and describes the software system product and lists developers or contractors and the related contact information as well as general comments regarding the project. The comment field allows the individual(s) entering information to describe additional relevant project information such as new functionality for this activity or that an Engineering Change Proposal (ECP) affected certain CSCIs. The Project Summary Data (Table I-3) description encompasses a set of project-related CSCIs.

Table I-3: Project summary data

Equiv CMMI Level	CSCI Status	Subcontr %	Estimating Tool	Env	Platform	Application	Reserved	Reserved
3	FQT	0	Sage	Mil	Space	Bus		

The CSCI information section lists each individual component by CSCI along with a description, maturity or status at the time of entry, and the reference model or tool used to originally develop the data.

The Environment section of the data entry form contains several subsections to capture the information relevant to the parameters associated with size, product, and development environment.

I.2 Software data collection details

The CSCI data entry forms (spreadsheet format) are model independent. In other words, independent of the model that was used in creating the project estimate data, the model definitions, phraseology, or methodology used to track the project data, the information will transfer accurately onto the data entry form. The parameter definitions include cross-references to like values in the major cost estimation models as well as the parameter Complexity Attribute (CA) definitions defined originally by Barry Holchin[85].

I.3 CSCI description

The following descriptions identify the data needed in each field and may include an example of the type of data to be entered:

[85] Barry Holchin, known for 1996 Code Growth Studies, developed Complexity Attributes for government contract software development data gathering efforts.

PROJ: Identifies the major acquisition program that contains the CSCI (e.g., MilStar).

CSCI ID: An alphanumeric name identifying the CSCI.

SUB CSCI ID: An alphanumeric designation identifying next level breakdown of the primary CSCI, if applicable.

VER: Identify the version number of the CSCI software being reported, i.e. v1.0, v2.0, Satellite 2-5, xyz, etc.

Contractor: The prime or sub-contractor involved in development of the CSCI, with point of contact information.

Description: The text description of the CSCI, the sub-CSCI functionality.

Equivalent CMMI Level: Configuration Maturity Model Integration (CMMI) assessment rating pertaining to the CSCI developer.

CSCI Status: Development stage of the CSCI.

Subcontractor %: Percentage of overall CSCI being developed by the subcontractor.

Estimating Tool: Estimating model (or estimating tool) utilized to develop the database CSCI data, (i.e., SEER™, Sage, Price, etc.).

Environment: Product taxonomy description for acquisition environment.

COM: Commercial acquisition environment description

GOV: Government acquisition environment description

MIL: Military acquisition environment description

Platform: Taxonomy descriptor for product platform

SPACE: Unmanned space platform

MANSPACE: Manned space platform

AVIONIC: Avionic space platform

MOBILE: Mobile software platform

GROUND: Fixed ground software platform

NAUTICAL: Nautical software platform

MANUFACTURING: Manufacturing system software platform

Application: Taxonomy descriptors for specific application areas.

PAYLOAD: Spacecraft payload

EXEC: Operation system/System executive

SUPPORT: Development and systems support applications

BUS: Spacecraft bus

OPSCTL: Operations and control application

PLAN: Missions planning application

TEST: Test and simulation software

TRAINING: User or maintenance training application

SIGNAL PROCESSING (SigProc): Signal processing application

COMMUNICATIONS: Communications software application

DATABASE: Database management/information system

MIS: Management information system

Reserved: Taxonomy identifiers not assigned.

I.3.1 Requirements

The CSCI information includes a description of the software requirements development activity; that is, a description of the work to be performed prior to the start of full-scale development. This is outside the effort used to determine the productivity rate, but is important in the computation of the overall development effort. The necessary elements are: the relative percentage of requirements completed before the start of the project, the formality of the requirements, and an indication of whether or not the software development effort includes the support effort following the start of full-scale development.

Table I-4: Requirements data

Requirements		
% Comp @ Start	Formality	Effort After Base
93	8%	no

% Comp @ Start: Amount of requirements development/specification (Table I-4) effort which will be complete at contract award.

Formality: Formality to which software requirements will be analyzed and specified. This parameter specifies the amount of effort added to the development estimate to support the definition prior to the start of requirements full-scale development. This effort includes definition, specification and review of software function, performance, interface, and verification requirements. The formality rating is specified as a percentage, typically in the range of 0-15% relative to the full-scale development effort.

Effort After Base: Software requirements that were supported after the up-front software requirements activity is complete.

I.3.2 Systems integration

Systems integration accounts for the effort expended to support the integration of the software product into a higher level system (Table I-5). A stand-alone software product has zero integration effort.

Table I-5: System integration data

System Integration		
Progs Concur	Concur of I&T Sched	Hardware Integ (%)
3	Nominal	28

Progs Concur: Number of CSCIs that will be concurrently integrated with the developed software and interface with it directly.

Concur of I&T Sched: Degree of concurrency or overlap between the development activities and the integration and testing activities. The rating is as follows:

Nominal	System integration occurs after CSCI testing is complete.
High	System integration begins during CSCI integration.
Very High	Majority of system integration occurs prior to CSCI development testing completion.
Extra High	All systems integration occurs during CSCI development.

Hardware Integ: Degree of difficulty in integrating the development software with the operational or target hardware. This parameter relates to the relative effort required in software-hardware integration. Ratings are specified as percentages, and are typically in the range of 0-32% relative to the full-scale software development effort. This effort is in addition to the

development effort (0% corresponds to no hardware integration, 32% corresponds to significant hardware integration effort with concurrent hardware development.)

I.4 Size data

I.4.1 Sizing data

The "Sizing Data" group of the data collection worksheet is segmented into major elements and further separated into more definitive sub-elements. Size data is employed to assemble the estimate for the CSCI development software effort. Software size can be specified as source lines of code (Table I-6) (SLOC) or as function points.

The major segments of the sizing data with appropriate sub-elements are as follows:

1. Source Code (KSLOC)
2. Reuse Adjustments
3. Software Source
4. Function Points
5. Programming Source Language
6. Requirements
7. System Integration

Table I-6: Source code sizes

Source Code (KSLOC)				
Total	**New**	**Reused**	**Modified**	**Effective**
45.9	3.9	42	0	7.6

I.4.1.1 Source code (KSLOC)

Total: Describes the total size of the program in thousands of source lines of code (KSLOC). This value relates to physical size characteristics and is determined by adding the new, modified and reused source code.

New: Specifies the number of source lines of code (KSLOC) to be created.

Reused: Total unmodified source code (KSLOC). Reused source code is alternately described as Verbatim or used without rework.

Modified: Specifies the amount of pre-existing source lines of code (KSLOC) at the module level to be modified in order to add or change an existing functionality in the developed CSCI. Modified code, by definition, is contained in reused (pre-existing) white-box modules. If one countable SLOC within a module is added, deleted, or modified, the entire countable SLOC in the module is counted as modified SLOC.

Effective (KESLOC): Measure of work required to produce a software task. Based on the number of KSLOC to be produced, the effort to understand the existing software testing and integration effort required.

I.4.1.2 Reuse adjustments

The reuse adjustments (Table I-7) are represented by %RD, %RI, and %RT.

Table I-7: Reuse data

Reuse Adjustments		
%RD	**%RI**	**%RT**
.50	.50	.75

%RD: The amount of redesign (software architecting and detailed design) that is required to make the pre-existing software functional within the software item. The redesign factor includes effort required to make: (1) software architecture changes, (2) software detailed design changes, and (3) documentation updates. This factor also includes reverse-engineering required to (1) understand the software prior to updates and deletions, and

(2) revalidate the software design. The value is specified as a decimal fraction.

%RI: The portion of the pre-existing code that requires re-implementation (coding and unit testing) to make it functional within the software item. The implementation factor includes effort required to: (1) evaluate interface definitions for changes in reused software, (2) code changes to the reused software, (3) formally review code changes, and (4) perform unit testing of code impacted by the changes to the implementation. The value is specified as a decimal fraction.

%RT: The effort required to test the pre-existing software, expressed as a portion of the effort that would have been required had the software been developed from scratch. The test factor includes effort required to: (1) update test plans and procedures, (2) develop test drivers and simulators, (3) perform formal integration testing, and (4) generate test reports for the updated reused software. The value is specified as a decimal fraction.

Table I-8: Reuse source

SW Source
Milstar

I.4.1.3 Software source

SW Source: For pre-existing code, name of original software system (Table I-8) that contributed the source code (e.g., Milstar).

I.4.1.4 Function points

Function points are an alternative measure for specifying the size (Table I-9) of a software system that quantifies the information processing functionality associated with major external data or control input, output or file types. *The counting guidelines are specified according to the definitions and counting rules published by the International Function Point Users Group (IFPUG) in the Function Point Counting Practices: Manual Release 4.2*.

Table I-9: Function Point Data

Function Points				
New	**Deleted**	**Modified**	**UFP**	**VAF**
50	3	7	40	1.2

UFP: Unadjusted function point count.

VAF: Value adjustment factor modifies the UFP count to account for software technical and operational characteristics.

I.4.1.5 Programming source language

The programming source language (Table I-10) is the primary programming language(s) used by the programmers in the development of the software product. The language name(s) should correspond to the source language count.

Program Source Lang: The implementation language for the software product.

Table I-10: Source language

Program Source Language
C++/Ada

I.5 Development environment data

The development effort and schedule data cannot be determined by the size data alone as described in Equation (I-1). The environment adjustment factor C_k accounts for all of the environmental effects present in the software development. The effects can be grouped into the following four categories:

1. Personnel
2. Support

3. Management
4. Product

The purpose of the CSCI environment (Attributes) data (Table I-11) is to capture pertinent, valid information relating to the economic and time expenditures on a project that translate into project costs. The availability of this project information allows the cost and schedule to be estimated in such a way as to reduce the risk of either over or under estimating effort and schedule.

The project environment factors are entered in the development environment section using the 0 to 10 attribute ratings described in Section I.8 (Development environment attributes) for each of the parameters.

The Attribute Ratings are generic descriptors for the various values associated with the particular model that the data was created or tracked in. Not all cells will have values.

Table I-11: Environment data

	Z	AA	AB	AC	AD	AE	AF	AG	AH	AI	AJ	AK	AL	AM	AN	AO	AP	AQ	AR	AS	AT	AU	AV	AW	AX	AY	AZ	BA	BB	BC	BD	BE	BF	BG	BH
3	Attribute Ratings (0-10)																																		
4	Personnel								Support								Management					Product													
5																																			
6	ACAP	PCAP	PROFAC	AEXP	DEXP	LEXP	PEXP	TEXP	DSYS	DVOL	MODP	PIMP	PVOL	RUSE	SCED	TOOL	MCLS	MORG	MULT	PCON	RLOC	CPLX	DATA	DISP	HOST	INTEGE	INTEGI	MEMC	PLAT	RELY	RTIM	RVOL	SECR	TIMC	TVOL
7	7	4		5	6	4	4				0	2	2		5	2			5			6	5	3	7	8	8	3		6	7	5	4	9	7
8	5	4		5	1	2	6				5	3	2	0	5	4			5			8	4	0	4	4	4	3		8	9	7	0	4	3

I.6 Cost, schedule data

The next data group to be collected is the effort (person-months), schedule and development technology constant (Table I-12). This data must be collected at the end of formal development; that is, the Final Qualification Test (FQT) associated with the customer acceptance of the CSCI. The data to be collected includes three major activities:

1. The effort expended prior to the start of full-scale development that corresponds to the acceptance of the SRS and ICD
2. The total full scale development effort including architecture design, code and unit test, and CSCI integration
3. The effort required for integration of the CSCI into the next higher assembly (Initial Operational Capability)

The schedule measures the elapsed time (months) from the start of full-scale development through FQT.

The development productivity is computed from the ratio of the effective SLOC (ESLOC) S_e and the full-scale development effort E_d; that is,

$$PR = S_e / E_d \text{ lines per person-month (ESLOC/PM)} \quad \text{(I-2)}$$

Productivity is often stated in hours per ESLOC or ESLOC per hour. The basis of the LPPM measure is generally assumed to be 152 hours per person-

month. A basis of 160 hours per person-month (PM) is sometimes used, but that occurrence is rare.

Table I-12: Development effort and schedule data

Current Effort (PM)				Current Schedule	
SLOC per PM	Req. Analysis (PM)	Full-Scale Devel (PM)	Integration Test (PM)	Elapsed Schedule (Months)	Milestone Status
125	242	1874	483	25.5	SDR

ESLOC per PM: Effective source lines of code per person-month are one of many productivity measures. This measure specifies the rate of ESLOC production.

Req Analysis: The number of months (PM) expended in performing requirements analysis prior to the start of full-scale development (Software Requirements Review).

Full-Scale Development: Total effort (PM) for analysis, development, and first level CSCI integration.

Integration Test: The number of months (PM) expended to perform integration testing.

Elapsed Schedule: The number of months elapsed since the beginning of the full-scale software development. The maximum elapsed schedule occurs at the end of the FQT.

Milestone Status: The point in the project life cycle that this information is obtained from (for example: SDR, PDR, CDR, FQT, etc.).

The values for requirements analysis, full-scale development, integration test, and elapsed schedule are actual measurements at FQT. The values at earlier milestones are projections or estimates. The important data at pre-FQT milestones is related to the project size, which tends to increase during development.

I.7 Technology constants

Technology constants (Table I-13) are measures of the efficiency, capability, or "goodness" of the development organization. The basic technology constant describes the developer's raw capacity unimpeded by the project constraints or project imposed environment. The effective technology constant includes the efficiency limitations imposed by the software development constraints and directly impacts the effort (cost), schedule, and productivity.

Basic Technology Constant (C_{tb}): The basic technology constant is typically between 5,500 and 7,500. The C_{tb} value is obtained from the Sage and/or SEER-SEM™ estimating tools or can be calculated from the development environment attributes. The basic technology constant is formally defined in Section 8.2.

Effective Technology Constant (C_{te}): The effective technology constant is the resulting technology measure that accounts for the degradation of the development team efficiency due to the limitations of the environment and product constraints. The effective technology constant is always less than

Table I-13: Technology constants

Technology Constants	
Basic	Effective
6,474.8	3,150.2

the basic technology constant. The effective technology constant is formally defined in Section 8.2.

I.8 Development environment attributes

The remainder of this appendix consists of tables presenting the various project environment factors which are entered in the development environment. The parameters are grouped by the four categories explained in Section I.5. Additional information may be applicable and, if so, is provided following the relevant rating value table.

Note that not all cells or rows will have values that apply to that factor.

I.8.1 Personnel

Detailed discussion of the personnel parameters (Tables I-14 through I-21) are contained in Sections 8 (Developer capability evaluation) and 9 (Development environment evaluation).

Table I-14: ACAP rating values

ACAP – Analyst Capability						
Value	**CA Definition 7**	**Sage**	**SEER-SEM**	**REVIC`**	**COCOMO II**	**PRICE-S**
0	Near Perfect/Perfect Functioning Team (90-100th percentile)	Highly motivated AND experienced team organization	Near Perfect Functioning Team (90th percentile)	90th percentile	90th percentile	Outstanding crew
1						
2	Extraordinary (80th percentile)	Highly motivated OR experienced team organization				Extensive experience
3			Extraordinary (75th percentile)	75th percentile	75th percentile	
4	Functional AND Effective (60th percentile)					
5	50th percentile	Traditional software development organization	Functional AND effective (55th percentile)	55th percentile	55th percentile	Normal crew
6						
7						Mixed experience
8	Functional with low effectiveness (30th percentile)	Poorly motivated OR non-associative organization	Functional with low affectivity (35th percentile	35th percentile	35th percentile	
9						Relatively inexperienced
10	Non-functioning team (5-15th percentile)	Poorly motivated AND non-associative organization	Poorly functioning or non-functioning team (5-15th percentile)	15th percentile	15th percentile	

Table I-15: PCAP rating values

PCAP – Programmer Capability						
Value	**CA Definition 31**	**Sage**	**SEER-SEM**	**REVIC`**	**COCOMO II**	**PRICE-S**
0	Near Perfect/Perfect Functioning Team (90-100th percentile)	Highly motivated AND experienced team organization	Near Perfect Functioning Team (90th percentile)	90th percentile	90th percentile	Outstanding crew
1						
2	Extraordinary (80th percentile)	Highly motivated OR experienced team organization				Extensive experience
3			Extraordinary (75th percentile)	75th percentile	75th percentile	
4	Functional AND Effective (60th percentile)					
5	50th percentile	Traditional software development organization	Functional AND effective (55th percentile)	55th percentile	55th percentile	Normal crew
6						
7						Mixed experience
8	Functional with low effectiveness (30th percentile)	Poorly motivated OR non-associative organization	Functional with low affectivity (35th percentile	35th percentile	35th percentile	
9						Relatively inexperienced
10	Non-functioning team (5-15th percentile)	Poorly motivated AND non-associative organization	Poorly functioning or non-functioning team (5-15th percentile)	15th percentile	15th percentile	

Table I-16: PROFAC rating values

PROFAC – Productivity Factor						
Value	**CA Definition 36**	**Sage**	**SEER-SEM**	**REVIC`**	**COCOMO II**	**PRICE-S**
0	More than four years average experience					14
1						13
2	Three years average experience					12
3						11

PROFAC – Productivity Factor						
Value	**CA Definition 36**	**Sage**	**SEER-SEM**	**REVIC`**	**COCOMO II**	**PRICE-S**
4	Two years average experience					10
5						9
6	One year average experience					8
7						7
8	Four months average experience					6
9						5
10	Less than four months average experience					4

Table I-17: AEXP rating values

AEXP – Application Experience						
Value	**CA Definition 5**	**Sage**	**SEER-SEM**	**REVIC`**	**COCOMO II**	**PRICE-S**
0	More than 10 years average experience or reimplementation by the same team	More than 10 years average experience or reimplementation by the same team	More than 10 years or reimplementation by the same team			
1						Familiar product
2	Seven years average experience	Seven years average experience				
3			Six years average experience	Six years average experience	Six years average experience	
4						
5	Three years average experience	Three years average experience	Three years average experience	Three years average experience	Three years average experience	Normal, new product
6						
7						
8	One year average experience	One year average experience	One year average experience	One year average experience	One year average experience	
9				Six months average experience	Six months average experience	New line of business
10	Less than four months experience	Less than four months experience	Less than four months experience	Two months average experience	Two months average experience	

Table I-18: DEXP rating values

DEXP – Development System Experience						
Value	**CA Definition 4**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	More than four years average experience	More than four years average experience		More than four years average experience	More than four years average experience	
1						
2	Three years average experience	Three years average experience	Three years average experience	Three years average experience	Three years average experience	
3						
4	Two years average experience	Two years average experience	Two years average experience	Two years average experience	Two years average experience	
5						
6	One year average experience	One year average experience	One year average experience	One year average experience	One year average experience	
7						
8	Four months average experience	Four months average experience	Four months average experience	Four months average experience	Four months average experience	
9						
10	Less than four months experience	Less than four months experience	Less than four months experience	Less than four months experience	Less than four months experience	

Table I-19: LEXP ratings values

LEXP – Programming Language Experience						
Value	**CA Definition 6**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	More than four years average experience	More than four years average experience		Six years average experience	Six years average experience	
1			Four years average experience			
2	Three years average experience	Three years average experience	Three years average experience	Three years average experience	Three years average experience	
3						
4	Two years average experience	Two years average experience	Two years average experience			
5						Normal
6	One year average experience	One year average experience	One year average experience	One year average experience	One year average experience	
7				Six months average experience	Six months average experience	
8	Four months average experience	Four months average experience	Four months average experience			
9						

LEXP – Programming Language Experience						
Value	**CA Definition 6**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
10	Less than four months experience	Less than four months experience	Less than four months experience	Two months average experience	Two months average experience	New language

Table I-20: PEXP rating values

PEXP – Practices and Methods Experience						
Value	**CA Definition 32**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	More than four years average experience	More than four years average experience		Six years average experience	Six years average experience	
1			Four years average experience			
2	Three years average experience	Three years average experience	Three years average experience	Three years average experience	Three years average experience	
3						
4	Two years average experience	Two years average experience	Two years average experience			
5						
6	One year average experience	One year average experience	One year average experience	One year average experience	One year average experience	
7				Six months average experience	Six months average experience	
8	Four months average experience	Four months average experience	Four months average experience			
9						
10	Less than four months experience	Less than four months experience	Less than four months experience	Two months average experience	Two months average experience	

Table I-21: TEXP rating values

TEXP – Target System Experience						
Value	**CA Definition 34**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	More than four years average experience	More than four years average experience				
1			Four years average experience			
2	Three years average experience	Three years average experience	Three years average experience			
3						

TEXP – Target System Experience						
Value	**CA Definition 34**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
4	Two years average experience	Two years average experience	Two years average experience			
5						
6	One year average experience	One year average experience	One year average experience			
7						
8	Four months average experience	Four months average experience	Four months average experience			
9						
10	Less than four months experience	Less than four months experience	Less than four months experience			

The Target System Experience (TEXP) is the *effective average* experience (years) of the development team with the target (final) system on which the developed software product will execute, including both the hardware environment and the resident operating system, if any. If the target system is essentially the same as the development system, then TEXP will be equal to the Development System Experience (DEXP).

I.8.2 Support

The Development System Complexity (DSYS) parameter (Table I-22) rates the relative complexity of the development system, compilers, file interfaces and support environment. This parameter is closely linked to DEXP.

Table I-22: DSYS rating values

DSYS – Development System Complexity						
Value	**CA Definition 35**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	Single module (no external interfaces)	Single-user machines (Windows, Mac), standalone systems, may be networked	Single user machines (Windows, Mac), standalone systems, may be networked			
1						
2	Loosely coupled items					
3						
4						
5	Minimum coupling and timing constraints	Multi-user systems (NT Server, VAX VMS, UNIX)	Multi-user systems (NT Server, VAX VMS, UNIX)			
6						
7	Numerous or complex interfaces					
8						
9						

DSYS – Development System Complexity						
Value	**CA Definition 35**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
10	Tight coupling and timing constraints	Distributed network where developers must have cognizance of the distributed functionality	Distributed network where developers must have cognizance of the distributed functionality			

Development System Volatility (DVOL) determines the impact of changes to the development virtual machine. These may be changes in the program editors, compilers or other tools, changes in the operating system and command languages, or changes in the development hardware itself.

- *REVIC* – DVOL (Table I-23) is contained in the Virtual Machine Volatility (VIRT) parameter. The target system volatility portion of VIRT is defined by the parameter Target System Volatility (TVOL).
- *COCOMO II* – The Platform Volatility Parameter (PVOL) refers to the complexity of hardware and software (operating system, database management system, etc.) the software product calls on to perform its tasks.

Table I-23: DVOL rating values

DVOL – Development System Volatility						
Value	**CA Definition 38**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	No major changes, minor changes each year	No major changes, minor changes each year	No major changes, minor changes each year			
1						
2	Major change each 12 months, minor each month	Annual major changes, minor monthly changes	Major change each 12 months, minor each month			
3						
4						
5	Major change each six months, minor each 2 weeks	Semi-annual major changes, minor bi-weekly changes	Major change each six months, minor each two weeks			
6						
7						
8	Major changes each two months, minor each week	Bi-monthly major changes, minor weekly changes	Major changes each two months, minor each week			
9						

DVOL – Development System Volatility						
Value	**CA Definition 38**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
10	Major change each two weeks, minor two times a week	Bi-weekly major changes, minor changes every two days	Major change each two weeks, minor two times a week			

The Modern Practices (MODP) (Table I-24) parameter evaluates the usage of modern software development practices and methods at the time the software design begins. The practices include analysis and design, structured or object-oriented methods, development practices for code implementation, documentation, verification and validation, database maintenance, and product configuration control. The use of modem software development methods is measured at the start of the formal development process (SRR) and includes only practices that can be considered organization culture at that time.

Table I-24: MODP use rating values

MODP – Modern Practices Use						
Value	**CA Definition 20**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	Maximum benefit of MPPs realized	SEI CMMI = 5	Routine use of a complete software development process	Routine use of a complete software development process		
1 2						
3	General use of MPPs by personnel experienced in their use	SEI CMMI = 4	Reasonably experienced in most practices	Reasonably experienced in most practices		
4						
5	Some use of MPPs by personnel experienced in their use	SEI CMMI = 3	Reasonably experienced in most practices	Reasonably experienced in most practices		
6 7						
8	Beginning experimental use of MPPs	SEI CMMI = 2	Beginning experimental use of practices	Beginning experimental use of practices		
9						
10	No use of MPPs	SEI CMMI = 1	No use of modern development practices	No use of modern development practices		

The Process Improvement (PIMP) (Table I-25) parameter evaluates the use of development technology improvement by comparing established (culture) development practices with those to be used in the current software development. The results are then compared with the SEI/CMMI ratings to measure the level of improvement planned.

Table I-25: Process improvement rating values

PIMP – Process Improvement						
Value	**CA Definition 33**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	No change in Modern Development Practices (MDP) use from the established development	No change in MDP use from the established development	No change in MDP(MDP) use from the established development			
1 2						
3	Moderate change – Organization improving development technologies equivalent to a 1 level CMMI transition on the MDP practices use rating	Moderate change – Organization improving development technologies equivalent to a 1 level CMMI transition on the MDP practices use rating	Moderate change – Organization improving development technologies equivalent to a 1 level CMMI transition on the MDP practices use rating			
4 5 6						
7	Major change – Organization improving development technologies equivalent to a 2 level CMMI transition on the MDP practices use rating	Major change – Organization improving development technologies equivalent to a 2 level CMMI transition on the MDP practices use rating	Major change – Organization improving development technologies equivalent to a 2 level CMMI transition on the MDP practices use rating			
8 9						
10	Extreme change – Organization improving development technologies equivalent to a 3 level CMMI transition on the MDP practices use	Extreme change – Organization improving development technologies equivalent to a 3 level CMMI transition on the MDP practices use	Extreme change – Organization improving development technologies equivalent to a 3 level CMMI transition on the MDP practices use			

The Practices and Methods Volatility (PVOL) (Table I-26) parameter rates the frequency of changes to the development practices and methods being used. This rating depends on the scope or magnitude of the changes, as well as the frequency with which they occur. A minor change is any change that impacts the development team, but does not require minor delays or

adjustments to the process. A major change requires a significant adjustment to the development process, and has a major impact on the development effort and schedule.

- *REVIC* – This attribute is accounted for in the VIRT parameter.

The reusability level required parameter measures the relative cost and schedule impact software reuse requirements. This factor depends on the breadth of the required reuse (RUSE) (Table I-27); for example, the relative cost of a product with single application is lower than that required for components designed for reuse across a broad spectrum of applications).

Table I-26: PVOL rating values

PVOL – Practices/Methods Volatility						
Value	**CA Definition 25**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	No major changes, minor changes each year	No major changes, minor changes each year	No major changes, minor changes each year	No major changes, minor changes each year		
1						
2	Major change each 12 months, minor each month	Annual major changes, minor monthly changes	Major change each 12 months, minor each month	Major change each 12 months, minor each month		
3						
4						
5	Major change each six months, minor each two weeks	Semi-annual major changes, minor bi-weekly changes	Major change each six months, minor each two weeks	Major change each six months, minor each two weeks		
6						
7						
8	Major changes each two months, minor each week	Bi-monthly major changes, minor weekly changes	Major changes each two months, minor each week	Major changes each two months, minor each week		
9						
10	Major change each two weeks, minor two times a week	Bi-weekly major changes, minor changes every two days	Major change each two weeks, minor two times a week	Major change each two weeks, minor two times a week		

Table I-27: RUSE rating values

RUSE – Reusability Level Required						
Value	**CA Definition 27**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	No reusability required	No reuse	No reusability required	No reuse	None	
1						
2						
3	Software designed for reuse within a single application area (single contractor; multiple/single customers)	Reuse single mission products	Software will be reused within a single application area	Reuse single mission products	Across project	
4						
5					Across program	
6						
7	Software designed for reuse within a single product line (multiple contractors; multiple/single customers)	Reuse across single product line	Software will be reused within a single product line. Reusability may impact multiple development teams	Reuse across single product line	Across product line	
8						
9						
10	Mission software developed with full reusability required. All components of the software must be reusable	Reuse in any application	Mission software developed with full reusability required. All components of the software must be reusable	Reuse in any application	Across multiple product lines	

The schedule expansion/compression parameter Schedule Constraint (SCED) (Table I-28) relates the proposed schedule to the optimum development schedule. Optimum is interpreted in terms of most efficient personnel utilization or efficiency. Minimum development schedule is the shortest development schedule possible with a defined size, complexity, and technology (personnel, tools, etc.).

- *Sage, SEER-SEM* – The SCED parameter cannot be used to compress the schedule, since Sage and SEER-SEM compute (by default) the minimum development time. The SCED parameter can be used to expand the schedule.

Table I-28: Required schedule rating values

SCED – Required Schedule (calendar months)						
Value	**CA Definition 2**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	Time to development completion; 75% of optimal development schedule	Minimum schedule	Minimum schedule	<75% use of available execution time	<75% use of available execution time	

SCED – Required Schedule (calendar months)						
Value	**CA Definition 2**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
1	80% of optimal completion date					
2	85% of optimal completion date			85% use of available execution time	85% use of available execution time	
3	90% of optimal completion date					
4	95% of optimal completion date					
5	Optimal development schedule (100%)	Optimum schedule	Optimum schedule	100% use of available execution time	100% use of available execution time	
6	115% of optimal completion date					
7	130% of optimal completion date			130% use of available execution time	130% use of available execution time	
8	145% of optimal completion date					
9	160% of optimal completion date			160% use of available execution time	160% use of available execution time	
10	Greater than 175% of optimal completion date					

The automated tool support parameter (TOOL) (Tables I-29 and I-30) indicates the degree to which the software development practices have been automated and will be used in the software development. Practices not considered to be part of the development culture are not considered. The following list of tools and criteria can be used to aid in selection of the appropriate value.

Table I-29: Automated tool support levels of automation

V-Low Level of Automation (1950s era toolset) PA=10	**Nominal Level of Automation (1970s era) PA=5**
Assembler	Multi-user operating system
Basic linker	Interactive source code editor
Basic batch debug aids	Database management system
High level language compiler	Basic database design aids
Macro assembler	Compound statement compiler
	Extend overlay linker
Low Level of Automation (1960s era) PA=7	Interactive text editor
Overlay linker	Simple program design language (PDL)
Batch source editor	Interactive debug aids
Basic library aids	Source language debugger
Basic database aids	Fault reporting system
Advanced batch debug aids	Basic program support library

	Source code control system
	Nominal Level of Automation (1970s era) PA=5
	Virtual memory operating system
	Extended program design language
High Level of Automation (1980s era) PA=3	**V-High Level of Automation (2000s era) PA=1**
CASE tools	Integrated application development environment
Basic graphical design aids	Integrated project support environment
Advanced text editor (word processor)	Visual programming tools
Implementation standards enforcer	Automated code structuring
Static source code analyzer	Automated metrics tools
Program flow and test case analyzer	Graphical User Interface (GUI) testing tools
Full program support library w/CM Aids	4GLS (Fourth-Generation Languages)
Full integrated documentation system	Code generators
Automated requirements specification & analysis	Screen generators
General purpose system simulators	
Extended design tools and graphics support	
Automated verification system	
Special purpose design support tools	
Relational Database Management System (RDBM)	

Table I-30: Automated tool use rating values

TOOL – Automated Tool Support						
Value	**CA Definition 17**	**Sage**	**SEER-SEM**	**REVIC`**	**COCOMO II**	**PRICE-S**
0	Automated full Ada APSE	Fully integrated environment	Advanced fully integrated tool set	Strong, mature, proactive lifecycle tools, well integrated with processes, methods, reuse	Strong, mature, proactive lifecycle tools, well integrated with processes, methods, reuse	
1						
2	Fully integrated application development environment		Modern, fully automated application development environment, including requirements, design, and test analyzers	Strong, mature lifecycle tools, moderately integrated	Strong, mature lifecycle tools, moderately integrated	
3		Moderately integrated environment				
4	Modern (Ada Min. APSE & design, requirements, or test tools)		Modern visual programming tools, automated CM, test analyzers plus requirements or design tools			

TOOL – Automated Tool Support						
Value	**CA Definition 17**	**Sage**	**SEER-SEM**	**REVIC`**	**COCOMO II**	**PRICE-S**
5		Extensive tools, little integration, basic maxi tools	Visual programming, CM tools and simple test tools	Basic lifecycle tools, moderately integrated	Basic lifecycle tools, moderately integrated	
6	Interactive (Programmer Work Bench)		Interactive (Programmer Work Bench)			
7				Simple, front end, back end CASE, little integration	Simple, front end, back end CASE, little integration	
8	Basic batch tools (370 OS type (compiler, editor))	Basic maxi tools, basic micro tools	Basic batch tools (compiler, editor)			
9						
10	Primitive tools (Bit switches, dumps)	Very few primitive tools	Primitive tools (Bit switches, dumps)	Edit, code, debug	Edit, code, debug	

I.8.3 Management

The management environment effects are discussed in detail in Section 9 (Development environment evaluation).

The Multiple Security Classification (MCLS) (Table I-31) parameter evaluates the impact of security requirements on the software development with imposed multiple levels of security classification

- *SEER-SEM* – The security impact is contained in the Multiple Development Sites (MULT) parameter.
- *REVIC* – The multiple classification level impact is contained in the DoD Security Classification security (SECU) parameter.

Table I-31: Multiple project classification levels rating values

MCLS – Multiple Classification Levels						
Value	**CA Definition 39**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0		Multiple compartment classifications		Unclassified project		
1						
2						
3		Compartmentalized and non-cleared personnel				
4						
5		Classified and unclassified personnel				
6						
7						

MCLS – Multiple Classification Levels						
Value	**CA Definition 39**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
8		Common security classification				
9						
10				Classified project		

The Multiple Organization (MORG) (Table I-32) parameter evaluates the impact of multiple development organizations on the development project. Multiple organizations always arise when mixing personnel from different contractors. Multiple organizations within a single organization are possible, and often appear due to organization rivalry. For example, software, system, and test engineering groups within a single organization function as separate contractors.

- *SEER-SEM* – The multiple organization impact is contained in the MULT parameter.
- *PRICE-S* – The multiple organization impact is contained in the management complexity (CPLXM) parameter.

Table I-32: Multiple development organization rating values

MORG – Multiple Development Organizations						
Value	**CA Definition 40**	**Sage**	**SEER-SEM**	**REVIC`**	**COCOMO II**	**PRICE-S**
0		Multiple contractors; single site				
1						
2						
3		Prime and subcontractor organization; single site				
4						
5						
6		Developer using personnel from other organizations				
7						
8						
9						
10		Single development organization				

The multiple development site (MULT) (Table I-33) parameter describes the physical separation between personnel in the software development team.

SEER-SEM – MULT; *COCOMO II, REVIC* – SITE. The MULT and SITE parameters describe the organizational and site diversity within the personnel developing the software product. This separation can be due to physical location, political boundaries, or even security issues. A

program being developed in a mixed-security level environment should be considered as multiple organizations. Advanced communications techniques, such as e-mail, Wide Area Networks, or teleconferencing can reduce the impact of physical separation, but will not negate it.

Table I-33: Multiple development sites rating values

MULT – Multiple Development Sites						
Value	**CA Definition 1**	**Sage**	**SEER-SEM**	**REVIC`**	**COCOMO II**	**PRICE-S**
0	Single site, single organization	All personnel at a single site with the same development area	Single site, single organization		Fully collocated	Single site, single organization
1						
2					Same building or complex	
3		Multiple sites within close proximity	Single site, multiple organizations			
4					Same city or metro area	
5	Two sites or several organizations					Multiple development sites
6					Multi-city or multi-company	
7	Two sites and several organizations	Multiple sites within one hour separation	Multiple sites, same general location, or mixed clearance levels			
8					Multi-city and multi-company	
9	Five or more sites or complex organization interfaces					
10	Five or more sites and complex organization interfaces	Multiple sites with greater than one hour separation	Multiple sites, located 50 miles or more apart, or international participation		International	Multiple sites, located 50 miles or more apart, or international participation

The personnel continuity (PCON) (Table I-34) parameter describes the software development team's average annual turnover rate at the commencement of full-scale software development. This evaluation is based on historic— not anticipated— turnover rates.

Table I-34: Personnel continuity rating values

PCON – Personnel Continuity						
Value	**CA Definition 41**	**Sage**	**SEER-SEM**	**REVIC`**	**COCOMO II**	**PRICE-S**
0					3% per year	
1						

PCON – Personnel Continuity						
Value	**CA Definition 41**	**Sage**	**SEER-SEM**	**REVIC`**	**COCOMO II**	**PRICE-S**
2						
3					6% per year	
4						
5					12% per year	
6						
7					24% per year	
8						
9						
10					48% per year	

I.8.4 Product

The product characteristics impacts are discussed in detail in Section 10 (Product characteristics evaluation).

The software complexity rating (CPLX) (Table I-35), also referred to as staffing complexity, is a rating of the software system's inherent difficulty to produce in terms of the rate at which staff can be added to a project. The complexity is affected by the instruction mix (as in PRICE-S), as well as the algorithmic complexity of the software product.

Table I-35: Product complexity (staffing) rating values

CPLX – Complexity (Staffing)						
Value	**CA Definition 11**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	Extremely simple S/W with simple code, simple input/output (I/O) and internal storage arrays	Extremely simple S/W with simple code, simple I/O, and internal storage arrays	Extremely simple S/W with simple code, simple I/O, and internal storage arrays	Extremely simple S/W with simple code, simple I/O, and internal storage arrays	Extremely simple S/W with simple code, simple I/O, and internal storage arrays	
1						User-defined/math
2	Computational efficiency has some impact on development effort	Low logical complexity, simple I/O and internal data storage	Low logical complexity, simple I/O and internal data storage	Low logical complexity, simple I/O and internal data storage	Low logical complexity, simple I/O and internal data storage	String manipulation
3						
4	New standalone systems developed on firm operating systems. Minimal interface problems	New standalone systems developed on firm operating systems. Minimal interface problems	New standalone systems developed on firm operating systems. Minimal interface problems	New standalone systems developed on firm operating systems. Minimal interface problems	New standalone systems developed on firm operating systems. Minimal interface problems	Data Management
5	Typical command and control programs	Typical command and control programs	Typical command and control programs	Typical command and control programs	Typical command and control programs	Online command

CPLX – Complexity (Staffing)						
Value	**CA Definition 11**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
6	Minor real-time processing, significant logical complexity, some changes to OS					
7		Significant logical complexity, perhaps requiring changes to development OS, minor real-time processing	Significant logical complexity, perhaps requiring changes to development OS, minor real-time processing	Significant logical complexity, perhaps requiring changes to development OS, minor real-time processing	Significant logical complexity, perhaps requiring changes to development OS, minor real-time processing	Real-time
8	Challenging response time requirements, new system with significant interface and interaction requirements	New systems with significant interfacing and interaction requirements in a larger system structure	New systems with significant interfacing and interaction requirements in a larger system structure	New systems with significant interfacing and interaction requirements in a larger system structure	New systems with significant interfacing and interaction requirements in a larger system structure	Operating systems/interactive commands
9						
10	Very large data processing volume in a short time, signal processing system with complex interfaces.	Development primarily using micro code for the application	Development primarily using micro code for the application	Development primarily using micro code for the application	Development primarily using micro code for the application	

The database size (Table I-36) parameter captures the effect that large data requirements have on the software product development. The rating is determined by calculating D/P, the ratio of bytes in the database to SLOC in the program.

Table I-36: Database size rating values

Database Size						
Value	**CA Definition 21**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	No database					
1	Database (DB) size/ SLOC = 5					
2	DB size/SLOC = 10. Data easily managed. Requirements/structure known.			D/P SLOC<10	D/P SLOC<10	
3						
4	DB size/SLOC = 30			10≤D/P<100	10≤D/P<100	

Database Size						
Value	**CA Definition 21**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
5	DB size/SLOC = 55. Nominal DB size. Not access bound nor other critical constraint					
6						
7	DB size/SLOC = 450			100≤D/P<1000	100≤D/P<1000	
8	DB size/SLOC = 1,000. High access/performance requirements			D/P≥1000	D/P≥1000	
9						
10						

The Special Display Requirements (DISP) (Table I-37) rating specifies the amount of effort to implement user interface and other display interactions that cannot be accounted for by product size alone.

Table I-37: DISP rating values

DISP – Special Display Requirements						
Value	**CA Definition 16**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	No displays	Simple I/O requirements	Simple input/outputs: batch programs			
1						
2						
3	A few simple displays					
4		User-friendly (extensive human engineering)	User-friendly; error recovery and menus, basic Windows GUI not controlled by the application			
5	User-friendly error recovery and menus, character based, window formats, color					
6						
7	Interactive: touch screens, light pens, mouse, etc. Controlled by the computer program (graphics based – 1990s)	Interactive (mechanical user interface)	Interactive: light pen, mouse, touch screen, windows, etc. Controlled by the software being developed			
8						
9	High human-in-the-loop dependencies. Many interactive displays, monitors, or status inputs					

DISP – Special Display Requirements						
Value	**CA Definition 16**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
10	Complex requirements. Computer-Aided Design/Computer-Aided Manufacturing (CAD/CAM) Solid Modeling. Many interactive displays or status outputs (e.g., real-time alarms)	Complex requirements with severe impact (CAD/CAM)	Complex: CAD/CAM, 3D solid modeling			

The software development re-hosting (HOST) (Table I-38) parameter evaluates the effort to convert the software product from the development system to the target system. This rating is not applied to projects devoted to "porting" software from one system to another.

Table I-38: Development re-hosting requirements rating values

HOST – Development Re-hosting						
Value	**CA Definition 19**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	No rehosting required	No rehosting, common language and system	No rehosting, same language and system			
1						
2						
3	Minor language or minor system change	Minor language or minor system change				
4			Minor language and system change			
5	Minor language and minor system change	Minor language and minor system change				
6						
7	Major language or major system change	Major language or major system change	Major language system change			
8						
9						
10	Major language and major system change	Major language and major system change	Major language and system change			

The Integration - External (INTEGE) (Table I-39) parameter is used by PRICE-S to describe the level of difficulty of integrating and testing the CSCIs at the system level.

Table I-39: INTEGE requirements rating values

INTEGE – External Integration						
Value	**CA Definition 30**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	Loosely coupled interfaces/minimum operational constraints					Loosely coupled interfaces/minimum operational constraints
1						
2						
3						
4						
5	Nominal coupling and constraints					Nominal coupling and constraints
6						
7						
8						
9						
10	Tightly coupled interfaces/strict operational constraints					Tightly coupled interfaces/strict operational constraints

The PRICE-S Integration - Internal (INTEGI) (Table I-40) parameter describes the level of effort involved in integrating and testing the software product components up to the CSCI level.

Table I-40: INTEGI requirements rating values

INTEGI – Internal Integration						
Value	**CA Definition 22**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	Loosely coupled/minimum constraints and interaction					Loosely coupled/minimum constraints and interaction
1						
2						
3						
4						
5	Typical/closely coupled interfaces/many interrupts					Typical/closely coupled interfaces/many interrupts
6						
7						
8						
9						
10	Tightly coupled, strict constraints					Tightly coupled, strict constraints

The target system Memory Constraint (MEMC) (Table I-41) rating evaluates the development impact of anticipated effort to reduce application storage requirements. This attribute is not simply a measure of memory reserve, but is intended to reflect the effort required by the software developers to reduce memory usage. Using 99 percent of available resources represents no constraint if no effort was required to conserve resources.

- *COCOMO II* – The memory constraint (STOR) is used in the

COCOMO family of estimating tools to represent the main storage constraint imposed on a software system or subsystem development.

Table I-41: Memory constraints rating values

MEMC – Target System Memory Constraints						
Value	**CA Definition 13**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	Greater than 50% reserve of memory available	No memory economy measures required	No memory constraints	Greater than 50% reserve of memory available	Greater than 50% reserve of memory available	No memory economy measures required
1	45% reserve available			45% reserve available	45% reserve available	
2	40% reserve available			40% reserve available	40% reserve available	
3	35% reserve available	Some overlay use or segmentation required	Some overlay use or segmentation required	35% reserve available	35% reserve available	Some overlay use or segmentation required
4	30% reserve available			30% reserve available	30% reserve available	
5	25% reserve available			25% reserve available	25% reserve available	
6	20% reserve available			20% reserve available	20% reserve available	
7	15% reserve available	Extensive overlay and/or segmentation required	Extensive overlay and/or segmentation required	15% reserve available	15% reserve available	Extensive overlay and/or segmentation required
8	10% reserve available			10% reserve available	10% reserve available	
9	7% reserve available			7% reserve available	7% reserve available	
10	Additional memory must be provided	Complex memory management economy measure	Complex memory management economy measure	Additional memory must be provided	Additional memory must be provided	Complex memory management economy measure

The platform (PLAT) (Table I-42) parameter describes the customer's requirements stemming from the planned operating environment. It is a measure of the portability, reliability, structuring, testing, and documentation required for acceptable contract performance.

Table I-42: Software platform rating values

PLAT – Platform						
Value	**CA Definition 42**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0				Ground systems		Ground systems
1						
2				Mil-spec ground systems		Mil-spec ground systems
3						
4				Unmanned		Unmanned

PLAT – Platform						
Value	**CA Definition 42**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
				airborne		airborne
5				Manned airborne		Manned airborne
6				Unmanned space		Unmanned space
7						
8				Manned space		Manned space
9						
10				Manned space		Manned space

The Required Software Reliability (RELY) (Table I-43) parameter is the measure of the extent to which the software product must perform its intended function over a period of time. If the effect of a software failure is only slight inconvenience, then RELY is very low. If a failure would risk human life, then RELY is very high. This parameter defines planned, or required, reliability; it is not related to the Requirements Volatility (RVOL) due to inadequate requirements definition and planning. This cost driver can be influenced by the requirement to develop software for reusability (see the description for RUSE).

Table I-43: RELY rating values

RELY – Required Software Reliability						
Value	**CA Definition 14**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	Failure results in loss to human life	Risk to human life	Risk to human life	Risk to human life	Risk to human life	
1						
2	High financial loss	High financial loss	High financial loss	High financial loss	High financial loss	
3						
4						
5	Moderate recoverable loss	Moderate, easily recoverable loss	Moderate, easily recoverable loss	Moderate, easily recoverable loss	Moderate, easily recoverable loss	
6						
7	Minor inconvenience	Low, easily recoverable loss	Low, easily recoverable loss	Low, easily recoverable loss	Low, easily recoverable loss	
8						
9						
10	No requirement	Slight inconvenience	Slight inconvenience	Slight inconvenience	Slight inconvenience	

The Real-time Operations (RTIM) (Table I-44) rating evaluates the impact of the fraction of the software product that interacts with the outside environment. Usually, consider these communications as driven by the external environment or clock. This fraction will be large in event-driven systems such as process controllers and interactive systems.

The real-time rating is related to system behavior, not execution speed. Speed is evaluated by the TIMC parameter.

Table I-44: RTIM requirements rating values

RTIM – Real-time Operations Requirements						
Value	**CA Definition 29**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	0% of source lines with real-time considerations	Less than 25% of source code devoted to real-time operations	0% of source lines with real-time considerations			
1 2 3						
4	25% of source lines with real-time considerations	Approximately 50% of source code devoted to real-time operations	25% of source lines with real-time considerations			
5 6						
7	75% of source lines with real-time considerations	Approximately 75% of source code devoted to real-time operations	75% of source lines with real-time considerations			
8 9						
10	100% of source lines with real-time considerations	Nearly 100% of source code devoted to real-time operations	100% of source lines with real-time considerations			

The system RVOL (Table I-45) parameter evaluates the expected frequency and scope of requirements changes after baseline (SRR). Changes include both major and minor perturbations.

Table I-45: System requirements volatility rating values

RVOL – System Requirements Volatility						
Value	**CA Definition 9/10**	**Sage**	**SEER-SEM**	**REVIC`**	**COCOMO II**	**PRICE-S**
0	No changes	Essentially no requirements changes	Essentially no requirements changes	No changes		No changes
1 2						
3	Very few changes expected	Familiar product, small noncritical redirections	Small noncritical redirections	Very few changes expected		
4						
5	Minor changes to requirements caused by design reviews or changing mission requirements	Known product, occasional moderate redirections	Occasional moderate redirections, typical for evolutionary software developments	Minor changes to requirements caused by design reviews or changing mission requirements		

RVOL – System Requirements Volatility						
Value	**CA Definition 9/10**	**Sage**	**SEER-SEM**	**REVIC`**	**COCOMO II**	**PRICE-S**
6			Evolutionary software development with significant user interface requirements			
7	Some significant changes expected (none late in development phase)	Technology exists, unfamiliar to developer		Some significant changes expected (none late in development phase)		Changing requirements
8			Frequent moderate & occasional major changes			
9						New hardware
10	Expect major changes occurring at different times in development phase	Technology is new, frequent major redirections	Frequent major changes	Expect major changes occurring at different times in development phase		Parallel hardware development

CPLX1 (PRICE-S) - Complexity (See Requirements Volatility [RVOL]).

The System Security Requirement (SECR) (Table I-46) parameter evaluates the development impact of application software security requirements. Apply this parameter only if the software product is required to implement the security requirement. The Common Criteria Evaluation Assurance Level (EAL) describes the mission assurance category assigned to the system and the level of assurance mandated in the controls (DoDD 8500.1 and DoDI 8500.2).

Table I-46: System security requirement rating values

SECR – System Security Requirement						
Value	**CA Definition 28**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
0	Class D: Minimal protection – no security	CC EAL0: No security requirements (OB Class D)	Class D: Minimal protection – no security			
1		CC EAL1: Functional test				
2	Class C1: Access limited. Based on system controls accountable to individual user or groups of users. Simple project specific password protection	CC EAL2: Structural test (OB Class C1)	Class C1: Access limited. Based on system controls accountable to individual user or groups of users. Simple project specific password			

SECR – System Security Requirement						
Value	**CA Definition 28**	**Sage**	**SEER-SEM**	**REVIC**	**COCOMO II**	**PRICE-S**
			protection			
3						
4	Class C2: Users individually accountable via login operations, auditing of security relevant events and resource isolation (typical VAX operating system such as Virtual Memory System).VMS).	CC EAL3: Methodical test and check (OB Class C2)	Class C2: Users individually accountable via login operations, auditing of security relevant events and resource isolation (typical VAX operating system such as Virtual Memory System).VMS).			
5						
6	Class B1: In addition to C2, data labeling and mandatory access control are present. Flaws identified by testing are removed (classified or financial transaction processing).	CC EAL4: Methodical design, test and review (OB Class B1)	Class B1: In addition to C2, data labeling and mandatory access control are present. Flaws identified by testing are removed (classified or financial transaction processing).			
7						
8	Class B2: System segregated into protection critical and non-protection critical elements. Overall system resistant to penetration (critical financial processing)	CC EAL5: Semiformal design and test (OB Class B2)	Class B2: System segregated into protection critical and non-protection critical elements. Overall system resistant to penetration (critical financial processing)			
9	Class B3: System excludes code not essential to security enforcement. Audit capability is strengthened. System almost completely resistant to protection.	CC EAL6: Semiformal verification, design and test (OB Class B3)	Class B3: System excludes code not essential to security enforcement. Audit capability is strengthened. System almost completely resistant to protection.			
10	Class A1: Security formally verified by mathematical proof (only a few known systems)	CC EAL7: Formal verification, design and test (OB Class A)	Class A1: Security formally verified by mathematical proof (only a few known systems)			

The system Central Processing Unit (CPU) timing constraint (TIMC) (Table I-47) rating evaluates the development impact of anticipated effort to reduce application response time requirements. TIMC evaluates the impact of limited processor capability and special measures needed to meet time related performance requirements by specifying the percentage of application software that must be developed with timing performance issues incorporated.

- *COCOMO II, REVIC* – The execution Time Constraint (TIME) parameter is used by the COCOMO family of estimating tools to specify the CPU timing constraint impact.
- *PRICE-S* – The resource utilization parameter UTIL specifies the fraction of available hardware cycle time, or total memory capacity. The parameter describes the extra effort needed to adapt software to operate within limited processor and memory capabilities.

Table I-47: CPU timing constraints rating values

TIMC – System CPU Timing Constraint						
Value	**CA Definition 12**	**Sage**	**SEER-SEM**	**REVIC`**	**COCOMO II**	**PRICE-S**
0	50% CPU power still available during maximum utilization	No CPU time constraints	No CPU time constraints	<50% of source code time constrained	<50% of source code time constrained	No CPU time constraints
1	45% CPU power still available during maximum utilization					
2	40% CPU power still available during maximum utilization					
3	35% CPU power still available during maximum utilization					
4	30% CPU power still available during maximum utilization	Approximately 25% of source code time constrained	Approximately 25% of source code time constrained	Approximately 70% of source code time constrained	Approximately 70% of source code time constrained	Approximately 25% of source code time constrained
5	25% CPU power still available during maximum utilization					
6	20% CPU power still available during maximum utilization					
7	15% CPU power still available during maximum utilization	Approximately 50% of source code time constrained	Approximately 50% of source code time constrained	Approximately 85% of source code time constrained	Approximately 85% of source code time constrained	Approximately 50% of source code time constrained
8	10% CPU power still available during maximum utilization					

TIMC – System CPU Timing Constraint						
Value	**CA Definition 12**	**Sage**	**SEER-SEM**	**REVIC`**	**COCOMO II**	**PRICE-S**
9	7% CPU power still available during maximum utilization					
10	5% CPU power still available during maximum utilization	Approximately 75% of source code time constrained	Approximately 75% of source code time constrained	Approximately 95% of source code time constrained	Approximately 95% of source code time constrained	Approximately 75% of source code time constrained

The target system volatility (TVOL) (Table I-48) parameter rates the anticipated stability (instability) created by changes and/or failures in the target system including hardware, tools, operating systems, and languages or compilers. A major change (or failure) requires redirection to continue the development. A minor change (or failure) allows workarounds to practices shortcomings and weaknesses.

Table I-48: TVOL rating values

TVOL – Target System Volatility						
Value	**CA Definition 3**	**Sage**	**SEER-SEM**	**REVIC`**	**COCOMO II**	**PRICE-S**
0	No hardware development	No major changes, annual minor changes	No major changes, minor changes each year			
1						
2						
3	Small amount of hardware development, localized impact on software	Annual major changes, monthly minor changes	Major change each 12 months, minor each month			
4						
5	Some overlaps of development. Most hardware available for testing software and vice versa	Semiannual major changes, biweekly minor changes	Major change each six months, minor each two weeks			
6						
7	Much overlap, little hardware available for testing	Bimonthly major changes, weekly minor changes	Major changes each two months, minor each week			
8						
9						
10	Simultaneous development, separate organizations, etc.	Biweekly major changes, minor changes every two days	Major change each two weeks, minor two times a week			