


Title	Software Engineering
Author	
Prof. Rajib Mall	
Acknowledgement	
The University acknowledges with thanks to NPTEL and the author for providing the study material on NPTEL portal under Creative Commons Attribution-NonCommercial-ShareAlike - CC-BY-NC-SA.	
	
This course on Software Engineering, available at NPTEL repository under Creative Commons Attribution-NonCommercial-ShareAlike - CC-BY-NC-SA, is adopted by Uttarakhand Open University for Master of Computer Applications program. All copyrights belong to their respective owners.	

Table of Contents

UNIT I: COMPUTATIONAL COMPLEXITY	1
1.0 LEARNING OBJECTIVES	1
1.1 Scope and Necessity of Software Engineering	1
1.1.1 Causes and solutions for software crisis	4
1.1.2 Program vs. software product	6
Check Your Progress I	6
1.2. STRUCTURED PROGRAMMING	6
1.2.1 Important features of a structured program	7
1.2.2 Advantages of structured programming	7
1.2.3 Exploratory style vs. modern style of software development	8
Check Your Progress II	10
1.3 SUMMARY	11
1.4 MODEL QUESTIONS	12
Answers to Check Your Progress I	12
Answers to Check Your Progress II	14
UNIT II: SOFTWARE LIFE CYCLE MODEL	15
2.0 LEARNING OBJECTIVES	15
2.1 BASICS OF SOFTWARE LIFE CYCLE AND WATERFALL MODEL	15
2.1.1 Life cycle model	15
2.1.2 The need for a software life cycle model	16
2.1.3 Different software life cycle models	16
2.1.4 Different phases of the classical waterfall model	17
2.1.4.1 Activities in each phase of the life cycle	18
4.1.4.2 Case Study	18
Check Your Progress I	23
2.2. PROTOTYPING AND SPIRAL LIFE CYCLE MODELS	24
2.2.1 Prototype	24
2.2.1.1 Need for a prototype in software development	24
2.2.1.2 Examples for prototype model	25
2.2.2 Spiral model	26
2.2.2.1 Circumstances to use spiral model	27
2.2.3 Comparison of different life-cycle models	27
Check your Progress II	28
2.3 SUMMARY	29
2.4 MODEL QUESTIONS	30
Answers to Check Your Progress I	31
Answers to Check your Progress II	32
UNIT III: REQUIREMENTS ANALYSIS AND SPECIFICATION	33
3.0 LEARNING OBJECTIVES	33
3.1 BASIC CONCEPTS IN REQUIREMENTS ANALYSIS AND SPECIFICATION	34
3.1.1 Role of a system analyst	34
3.1.2 Parts of a SRS document	35
3.1.3 Identifying functional requirements from a problem description	36
3.1.4 Problems without a SRS document	38
3.1.5 Identifying non-functional requirements	39
3.1.6 Decision tree	39
3.1.7 Decision table	40
Check Your Progress I	41
3.2. FORMAL REQUIREMENTS SPECIFICATION	42
3.2.1 Formal technique	42
3.2.1.1 Formal specification language	42
3.2.2 Model-oriented vs. property-oriented approaches	43

3.2.3 Branching Semantics	45
3.2.4 Axiomatic specification	47
3.3 ALGEBRAIC SPECIFICATION	48
3.3.1 Representation of algebraic specification	49
3.3.2 Properties of algebraic specifications	52
3.3.3 Structured specification	52
3.3.4 Advantages and disadvantages of algebraic specifications	53
3.3.5 Executable specification language (4GLs)	53
Check Your Progress II.....	53
3.4 SUMMARY	54
3.6 MODEL QUESTIONS.....	55
Answers to Check Your Progress I.....	56
Answers to Check Your Progress II.....	56
UNIT IV: SOFTWARE DESIGN ISSUES	57
4.0 LEARNING OBJECTIVES	57
4.1 BASIC CONCEPTS IN SOFTWARE DESIGN	57
4.1.1 Software design and its activities.....	57
Check Your Progress I.....	60
4.2. AN OVERVIEW OF CURRENT DESIGN APPROACHES	61
4.2.1 Cohesion	61
4.2.1.1 Classification of cohesion.....	61
4.2.2 Coupling	62
4.2.2.1 Classification of Coupling	63
4.2.3 Functional independence.....	63
4.2.3.1 Need for functional independence	64
4.2.3.2 Function-oriented design.....	64
4.2.3.3 Object-oriented design	65
4.2.3.4 Function-oriented vs. object-oriented design approach	65
4.2.3.5 Example: Fire-Alarm System	66
Check Your Progress II.....	68
4.3 SUMMARY	68
4.4 MODEL QUESTIONS.....	70
Answers to Check Your Progress I.....	71
Answers to Check Your Progress II.....	72
UNIT V: FUNCTION-ORIENTED SOFTWARE DESIGN	73
5.0 LEARNING OBJECTIVES.....	73
5.1 DATA FLOW DIAGRAMS (DFDS)	73
5.1.1 Structured Analysis	73
5.1.2 Data Flow Diagram (DFD).....	74
5.1.2.1 Importance of DFDs in a good software design	75
5.1.3 Data dictionary.....	75
5.1.3.1 Importance of data dictionary	77
5.1.4 Balancing a DFD	78
Check Your Progress I.....	78
5.2 DFD MODEL OF A SYSTEM	79
5.2.1 Context diagram.....	79
5.2.2 DFD model of a system	81
5.2.3 Data dictionary for a DFD model.....	84
5.2.4 Commonly made errors while constructing a DFD model	88
5.2.5 Shortcomings of a DFD model.....	90
Check Your Progress II.....	90
5.3 SUMMARY	91
5.4 MODEL QUESTIONS	92
Answers to Check Your Progress I.....	93
Answers to Check Your Progress II.....	94

UNIT VI: BASIC CONCEPTS IN OBJECT ORIENTATION	95
6.0 LEARNING OBJECTIVES.....	95
6.1. STRUCTURED DESIGN.....	95
6.1.1 Structured Design.....	95
6.1.1.1 Structure Chart.....	95
6.1.1.2 Structure Chart vs. Flow Chart	96
6.2 Transform Analysis.....	96
Check Your Progress I.....	98
6.2.1 Transaction Analysis.....	99
Check Your Progress II.....	100
6.3 SUMMARY.....	101
6.4 Model Question	101
Answers to Check Your Progress I.....	102
Answers to Check Your Progress II.....	102
UNIT VII: OBJECT MODELING USING UML	103
7.0 LEARNING OBJECTIVES.....	103
7.1. BASIC IDEAS ON UML.....	104
7.1.1 Model.....	104
7.1.1.1 Need for a model	104
7.1.2 Unified Modeling Language (UML)	105
7.1.2.1 Origin of UML.....	105
7.1.2.2 UML diagrams	106
7.2 USE CASE MODEL.....	107
7.2.1 Purpose of use cases	108
7.2.2 Representation of use cases	108
7.2.3 Utility of use case diagrams	111
7.2.4 Factoring of use cases	111
7.2.5 Organization of use cases	114
Check Your Progress I.....	115
7.3 CLASS AND INTERACTION DIAGRAMS.....	116
7.3.1 Class diagrams.....	116
7.3.2 Association vs. Aggregation vs. Composition	119
7.3.3 Inheritance vs. Aggregation/Composition.....	120
7.3.4 Interaction Diagrams.....	122
7.3.4.1 Sequence Diagram	123
7.3.4.2 Collaboration Diagram	124
7.4 ACTIVITY AND STATE CHART DIAGRAM	125
7.4.1 Activity diagrams.....	125
7.4.2 Activity diagrams vs. procedural flow charts	126
7.4.3 State chart diagram.....	127
7.4.4 Activity diagram vs. State chart diagram	128
Check Your Progress II.....	129
7.5 SUMMARY.....	129
7.6 MODEL QUESTIONS.....	131
Answers to Check Your Progress I.....	132
Answers to Check Your Progress II.....	132
UNIT VIII: OBJECT-ORIENTED SOFTWARE DEVELOPMENT.....	133
8.0 LEARNING OBJECTIVES.....	133
8.1 DESIGN PATTERNS.....	134
Check Your Progress I.....	138
8.2 DOMAIN MODELING	139
8.2.1 Booch's Object Identification Method	141
Check Your Progress II.....	151
8.3 SUMMARY.....	153
8.4 MODEL QUESTIONS.....	155

Answers to Check Your Progress I.....	156
Answers to Check Your Progress II.....	156
UNIT IX: USER INTERFACE DESIGN	157
9.0 LEARNING OBJECTIVES.....	157
9.1. BASIC CONCEPTS IN USER INTERFACE DESIGN	158
9.1.1 User guidance and online help.....	162
9.1.2 Mode-based interface vs. modeless interface	163
9.1.3 Graphical User Interface vs. Text-based User Interface.....	164
Check Your Progress I.....	164
9.2 TYPES OF USER INTERFACES	165
9.2.1 Command Language-based Interface	165
9.2.2 Direct Manipulation Interfaces	166
9.2.3 Menu-based Interface.....	166
9.2.4 Types of menus and their features	170
9.3 COMPONENT-BASED GUI DEVELOPMENT	171
9.3.1 Window	171
9.3.2 Window Management System (WMS).....	172
9.3.3 Window Manager and Window System	173
9.3.3.1 Window Manager	174
9.3.4 X-Window.....	175
9.3.4.1 Popularity of X-Window	175
9.3.4.2 Architecture of an X-System	176
9.3.5 Visual Programming	178
9.3.6 GUI design methodology.....	180
Check Your Progress II.....	183
9.4 SUMMARY.....	184
9.5 MODEL QUESTIONS	185
UNIT X: CODING AND TESTING	187
10.0 LEARNING OBJECTIVES.....	187
10.1 CODE REVIEW.....	188
10.1.1 Coding	188
10.1.1.1 Coding standards and guidelines	189
10.1.2 Code review	191
10.1.2.1 Code Walk Throughs	191
10.1.2.2 Code Inspection	192
10.1.3 Clean room testing	193
10.1.4 Software documentation	194
10.1.5 Program Testing.....	195
10.1.5.1 Aim of testing.....	196
10.2 BLACK-BOX TESTING	197
10.2.1 Testing in the large vs. testing in the small	197
10.2.1.1 Unit testing.....	197
10.2.1.2 Black box testing	198
10.2.2 Equivalence Class Partitioning	199
10.2.3 Boundary Value Analysis.....	200
Check Your Progress I.....	200
10.3 WHITE-BOX TESTING.....	201
10.3.1 Statement coverage.....	202
10.3.2 Branch coverage.....	203
10.3.3 Condition coverage	203
10.3.4 Path coverage	203
10.3.5 Control Flow Graph (CFG)	204
10.3.6 Cyclomatic complexity	206
10.3.7 Mutation testing	207
10.4 DEBUGGING, INTEGRATION AND SYSTEM TESTING	208

10.4.1	Need for debugging	208
10.4.2	Debugging approaches.....	208
10.4.2.1	Brute Force Method.....	208
10.4.2.2	Backtracking.....	209
10.4.2.3	Cause Elimination Method.....	209
10.4.2.4	Program Slicing	209
10.4.3	Debugging guidelines.....	209
10.4.4	Program analysis tools	210
10.4.5	Static program analysis tools	210
10.4.6	Dynamic program analysis tools	211
10.4.7	Integration testing	211
10.4.8	Integration test approaches.....	212
10.4.8.1	Big-Bang Integration Testing.....	212
10.4.8.2	Bottom-Up Integration Testing.....	212
10.4.8.3	Top-Down Integration Testing	213
10.4.8.4	Mixed Integration Testing.....	213
10.4.9	Phased vs. incremental testing	213
10.4.10	System testing.....	214
10.4.11	Performance testing.....	214
10.4.11.1	Stress Testing	215
10.4.11.2	Volume Testing	215
10.4.11.3	Configuration Testing.....	215
10.4.11.4	Compatibility Testing	216
10.4.11.5	Regression Testing	216
10.4.11.6	Recovery Testing	216
10.4.11.7	Maintenance Testing.....	216
10.4.11.8	Documentation Testing.....	217
10.4.11.9	Usability Testing	217
10.4.11.10	Error seeding.....	217
10.4.11.11	Regression testing.....	218
	Check Your Progress II.....	218
10.5	MODEL QUESTIONS	219
	UNIT XI: SOFTWARE PROJECT PLANNING	223
11.0	LEARNING OBJECTIVES.....	223
11.1	Project Planning and Project Estimation Techniques	224
11.1.1	Responsibilities of a software project manager.....	224
11.1.2	Project planning	225
11.1.3	Precedence ordering among project planning activities	225
11.1.4	Sliding Window Planning.....	226
11.1.5	Software Project Management Plan (SPMP)	226
11.1.6	Metrics for software project size estimation	228
11.1.6.1	Lines of Code (LOC)	228
11.1.6.2	Function point (FP).....	229
11.1.6.3	Feature point metric	232
11.1.7	Project Estimation techniques	233
11.1.7.1	Empirical Estimation Techniques	233
11.1.7.2	Heuristic Techniques.....	235
11.1.7.3	Analytical Estimation Techniques	236
11.2	COCOMO MODEL.....	240
11.2.1	Organic, Semidetached and Embedded software projects.....	240
11.2.2	COCOMO	241
	Check Your Progress I.....	246
11.3	Staffing Level Estimation and Scheduling	247
11.3.1	Staffing level estimation	247
11.3.1.1	Norden's Work	247

11.3.1.2 Putnam’s Work.....	248
11.3.2 Effect of schedule change on cost	249
11.3.3 Project scheduling.....	250
Check Your Progress II.....	256
11.4 MODEL QUESTIONS	257
UNIT XII: SOFTWARE PROJECT MONITORING AND CONTROL	261
12.0 LEARNING OBJECTIVES.....	261
12.1. ORGANIZATION AND TEAM STRUCTURES	262
12.1.1 Organization structure	262
12.1.1.1 Functional format vs. project format	262
12.1.1.2 Advantages of functional organization over project organization.....	263
12.1.1.3 Unsuitability of functional format in small organizations	264
12.1.2 Team structures	265
12.1.2.1 Chief Programmer Team	265
12.1.2.2 Democratic Team	266
12.1.2.3 Mixed Control Team Organization	267
12.1.2.4 Egoless programming technique.....	267
12.1.3 Characteristics of a good software engineer	268
Check Your Progress I.....	269
12.2. RISK MANAGEMENT AND SOFTWARE CONFIGURATION MANAGEMENT	270
12.2.1 Risk management.....	271
12.2.1.1 Risk Assessment	271
12.2.1.2 Risk containment	272
12.2.1.3 Risk leverage	272
12.2.1.4 Risk related to schedule slippage.....	272
12.2.2 Software configuration management.....	273
12.2.2.1 Release vs. Version vs. Revision	273
12.2.2.2 Necessity of software configuration management.....	274
12.2.2.3 Software configuration management activities.....	275
12.2.2.4 Configuration management tools	278
Check Your Progress II.....	278
12.3 MODEL QUESTIONS.....	280
UNIT XIII: SOFTWARE RELIABILITY AND QUALITY MANAGEMENT.....	283
13.1 SOFTWARE RELIABILITY ISSUES.....	284
13.1.1 Repeatable vs. non-repeatable software development organization	284
13.1.2 Software reliability	285
13.1.2.1 Reasons for software reliability being difficult to measure	285
13.1.2.2 Hardware reliability vs. software reliability differ	286
13.1.3 Reliability metrics.....	287
13.1.4 Classification of software failures	289
13.1.5 Reliability growth models	289
13.1.5.1 Jelinski and Moranda Model.....	289
13.1.5.2 Littlewood and Verall’s Model	290
13.2 STATISTICAL TESTING AND SOFTWARE QUALITY MANAGEMENT	290
13.2.1 Statistical testing.....	291
13.2.1.1 Operation profile.....	291
13.2.1.2 Steps in statistical testing.....	291
13.2.1.3 Advantages and disadvantages of statistical testing	291
13.2.2 Software Quality	292
13.2.2.1 Software quality management system	293
13.2.2.2 Evolution of quality management system	293
13.3 ISO 9000	294
13.3.1 ISO 9000 certification.....	294
13.3.2 Types of ISO 9000 quality standards.....	295
13.3.3 Software products vs. other products	295

13.3.4 Need for obtaining ISO 9000 certification	296
13.3.5 SUMMARY of ISO 9001 certification	296
13.3.6 Salient features of ISO 9001 certification	299
13.3.7 Shortcomings of ISO 9000 certification.....	299
13.4 SEI CMM	300
13.4.1 SEI Capability Maturity Model	300
13.4.2 Six Sigma	305
13.5 MODEL QUESTIONS.....	306
UNIT XIV: SOFTWARE MAINTENANCE.....	313
14.0 LEARNING OBJECTIVES.....	313
14.1 CHARACTERISTICS OF SOFTWARE MAINTENANCE	313
14.1.1 Necessity of software maintenance.....	313
14.1.2 Types of software maintenance.....	314
14.2 PROBLEMS ASSOCIATED WITH SOFTWARE MAINTENANCE	314
14.3 SOFTWARE REVERSE ENGINEERING.....	315
14.4 LEGACY SOFTWARE PRODUCTS	316
Check Your Progress I.....	316
14.5 FACTORS ON WHICH SOFTWARE MAINTENANCE ACTIVITIES DEPENDS	318
14.6 SOFTWARE MAINTENANCE PROCESS MODELS	318
14.7 SOFTWARE REENGINEERING.....	320
14.8 ESTIMATION OF APPROXIMATE MAINTENANCE COST	321
Check Your Progress II.....	321
14.9 MODEL QUESTIONS.....	322
UNIT XV: COMPUTER AIDED SOFTWARE ENGINEERING	325
15.0 LEARNING OBJECTIVES.....	325
15.1 BASIC IDEAS ON CASE TOOLS.....	326
15.1.1 CASE tool and its scope.....	326
15.1.1.1 Reasons for using CASE tools	326
15.1.1.2 CASE environment	326
15.1.1.3 CASE environment vs programming environment.....	327
15.1.1.4 Benefits of CASE	327
15.1.2 Requirements of a prototyping CASE tool	328
15.1.2.1 Features of a good prototyping CASE tool	328
15.1.2.2 Structured analysis and design with CASE tools	329
15.1.2.3 Code generation and CASE tools.....	329
15.1.2.4 Test case generation CASE tool.....	330
Check Your Progress I.....	330
15.2 DIFFERENT CHARACTERISTICS OF CASE TOOLS.....	331
15.2.1 Hardware and environmental requirements	331
15.2.2 Architecture of a CASE environment	333
15.2.2.2 Object Management System (OMS) and Repository	334
Check Your Progress II.....	334
15.3 MODEL QUESTIONS.....	335
UNIT XVI: SOFTWARE REUSE.....	337
16.0 LEARNING OBJECTIVES.....	337
16.1 BASIC IDEAS ON SOFTWARE REUSE	337
16.1.1 Advantages of software reuse	337
16.1.2 Artifacts that can be reused.....	338
16.1.3 Pros and cons of knowledge reuse	338
16.1.4 Easiness of reuse of mathematical functions.....	338
16.1.5 Basic issues in any reuse program	339
16.1.6 Domain analysis	340
Check Your Progress I.....	341
16.2 REUSE APPROACH	342
16.2.1 Components classification	342

16.2.2 Searching.....	343
16.2.3 Repository maintenance	344
16.2.4 Application generator	344
16.2.4.1 Advantages of application generators	344
16.2.4.2 Shortcomings of application generator	344
16.2.5 Re-use at organization level.....	345
16.2.6 Factors that inhibit an effective reuse program	347
Check Your Progress II.....	347
16.3 MODEL QUESTIONS.....	348
UNIT XVII: CLIENT-SERVER SOFTWARE DEVELOPMENT	349
17.0 LEARNING OBJECTIVES.....	349
17.1 BASIC IDEAS ON CLIENT-SERVER SOFTWARE DEVELOPMENT AND CLIENT-SERVER ARCHITECTURE	350
17.1.1 Client-server software.....	350
17.1.1.1 Advantages of client-server software	350
17.1.1.2 Factors for feasibility and popularity of client-server solutions	351
17.1.1.3 Advantages of client-server software development.....	351
17.1.1.4 Disadvantages of client-server software.....	353
17.1.1.5 Host-slave computing vs. client-server computing	354
17.1.2 Two-tier client-server architecture	354
17.1.2.1 Limitations of two-tier client-server architecture.....	354
17.1.3 Three-tier client-server architecture.....	355
17.1.3.1 Functions of middleware	356
17.1.3.2 Popular middleware standards	356
Check Your Progress I.....	356
17.2 CORBA AND COM/DCOM.....	357
17.2.1 Common Object Request Broker Architecture (CORBA).....	357
17.2.1.1 CORBA reference model	358
17.2.1.2 Explain CORBA architecture	359
17.2.1.3 Functions of Object Request Broker (ORB)	359
17.2.1.4 Commercial ORBs.....	360
17.2.1.5 Steps to develop application in CORBA.....	361
17.2.1.6 Dynamic Invocation Interface (DII) in CORBA	362
17.2.2 Component Object Model (COM)	362
17.2.3 Distributed Component Object Model (DCOM).....	362
17.2.4 Inter-ORB communication	362
17.2.5 Features of General Inter-ORB Protocol (GIOP).....	363
17.2.6 CORBA vs. COM/DCOM.....	363
Check Your Progress II.....	364
17.3 MODEL QUESTIONS.....	365
REFERENCE.....	367

UNIT I: COMPUTATIONAL COMPLEXITY

1.0 LEARNING OBJECTIVES

At the end of this lesson, the student will be able to:

- Identify the scope and necessity of software engineering.
- Identify the causes of and solutions for software crisis.
- Differentiate a piece of program from a software product.
- Identify the important features of a structured program.
- Identify the important advantages of structured programming over unstructured ones.
- Explain how software design techniques have evolved over the last 50 years.
- Differentiate between exploratory style and modern style of software development.

1.1 Scope and Necessity of Software Engineering

Software engineering is an engineering approach for software development. We can alternatively view it as a systematic collection of past experience. The experience is arranged in the form of methodologies and guidelines. A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are indispensable to achieve a good quality software cost effectively. These definitions can be elaborated with the help of a building construction analogy.

Suppose you have a friend who asked you to build a small wall as shown in fig. 1.1. You would be able to do that using your common sense. You will get building materials like bricks; cement etc. and you will then build the wall.



Fig. 1.1: A Small Wall

But what would happen if the same friend asked you to build a large multi-storeyed building as shown in fig. 1.2.



Fig. 1.2: A Multistoried Building

You don't have a very good idea about building such a huge complex. It would be very difficult to extend your idea about a small wall construction into constructing a large building. Even if you tried to build a large building, it would collapse because you would not have the requisite knowledge about the strength of materials, testing, planning, architectural design, etc. Building a small wall and building a large building are entirely different ball games. You can use your intuition and still be successful in building a small wall, but building a large building requires knowledge of civil, architectural and other engineering principles.

Without using software engineering principles, it would be difficult to develop large programs. In industry it is usually needed to develop large programs to accommodate multiple functions. A problem with developing such large commercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes as shown in fig. 1.3. For example, a program of size 1,000 lines of code has some complexity. But a program with 10,000 LOC is not just 10 times more difficult to develop, but may as well turn out to be 100 times more difficult unless software engineering principles are used. In such situations software engineering techniques come to rescue. Software engineering helps to reduce the programming complexity. Software engineering

principles use two important techniques to reduce problem complexity: abstraction and decomposition.

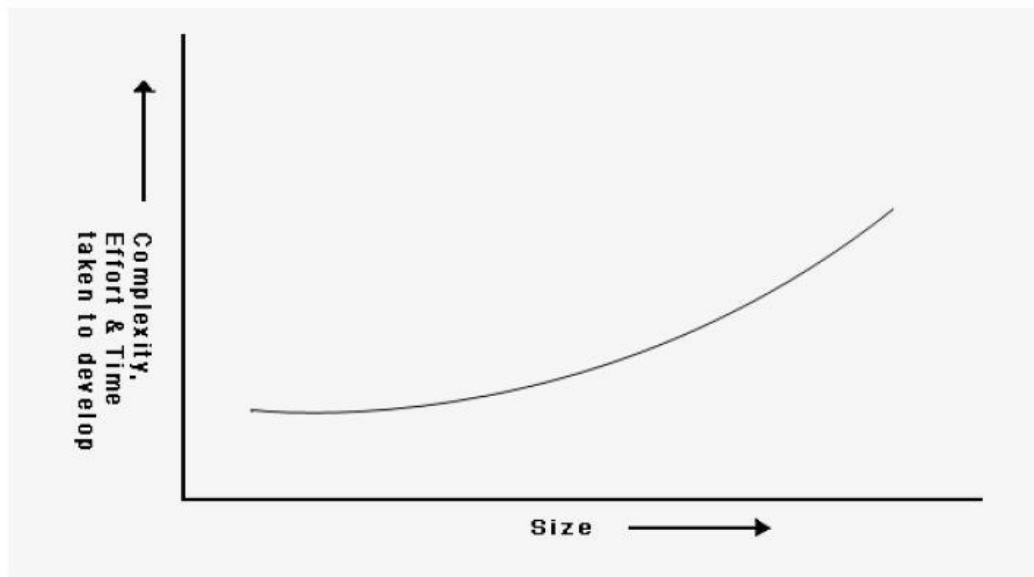


Fig. 1.3: Increase in development time and effort with problem size

The principle of abstraction (in fig.1.4) implies that a problem can be simplified by omitting irrelevant details. In other words, the main purpose of abstraction is to consider only those aspects of the problem that are relevant for certain purpose and suppress other aspects that are not relevant for the given purpose. Once the simpler problem is solved, then the omitted details can be taken into consideration to solve the next lower level abstraction, and so on. Abstraction is a powerful way of reducing the complexity of the problem.

The other approach to tackle problem complexity is decomposition. In this technique, a complex problem is divided into several smaller problems and then the smaller problems are solved one by one. However, in this technique any random decomposition of a problem into smaller parts will not help. The problem has to be decomposed such that each component of the decomposed problem can be solved independently and then the solution of the different components can be combined to get the full solution. A good decomposition of a problem as shown in fig.1.5 should minimize interactions among various components. If the different

subcomponents are interrelated, then the different components cannot be solved separately and the desired reduction in complexity will not be realized.

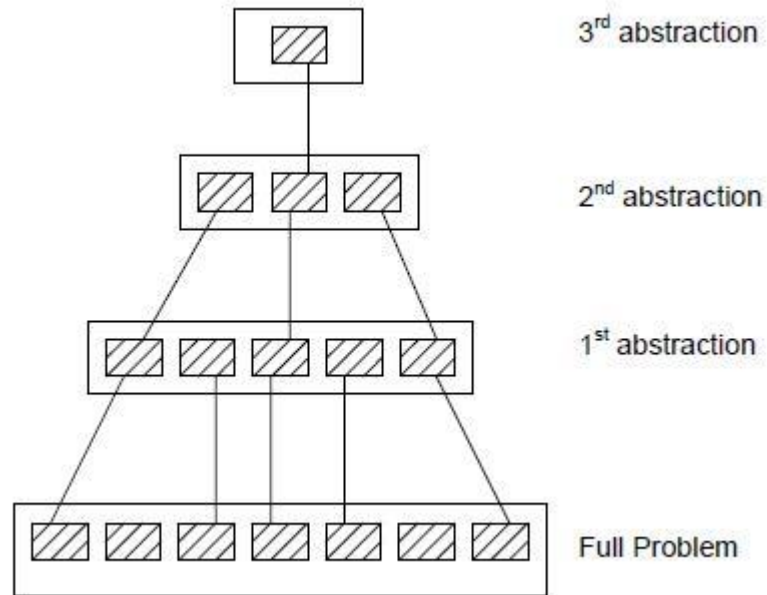


Fig. 1.4: A hierarchy of abstraction

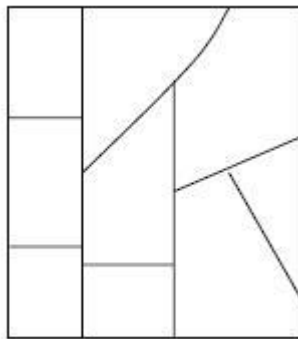


Fig. 1.5: Decomposition of a large problem into a set of smaller problems.

1.1.1 Causes and solutions for software crisis

Software engineering appears to be among the few options available to tackle the present software crisis.

To explain the present software crisis in simple words, consider the following. The expenses that organizations all around the world are incurring on software purchases compared to those on hardware purchases have been showing a worrying trend over the years (as shown in fig. 1.6)

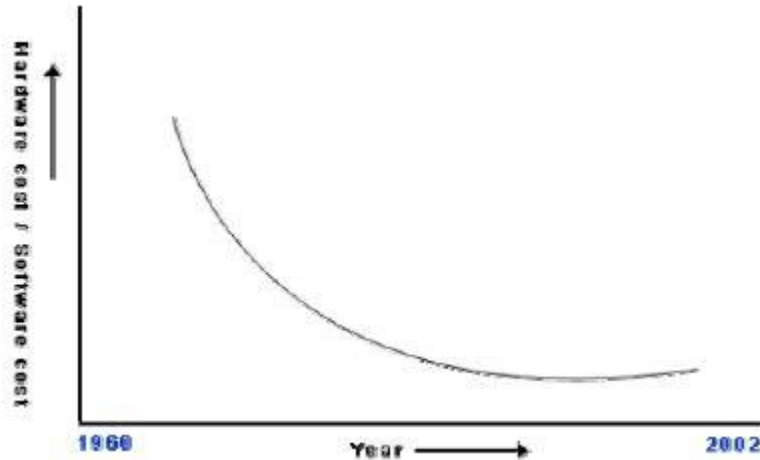


Fig. 1.6: Change in the relative cost of hardware and software over time

Organizations are spending larger and larger portions of their budget on software. Not only are the software products turning out to be more expensive than hardware, but they also present a host of other problems to the customers: software products are difficult to alter, debug, and enhance; use resources non-optimally; often fail to meet the user requirements; are far from being reliable; frequently crash; and are often delivered late. Among these, the trend of increasing software costs is probably the most important symptom of the present software crisis.

Remember that the cost we are talking of here is not on account of increased features, but due to ineffective development of the product characterized by inefficient resource usage, and time and cost over-runs.

There are many factors that have contributed to the making of the present software crisis. Factors are larger problem sizes, lack of adequate training in software engineering, increasing skill shortage, and low productivity improvements.

It is believed that the only satisfactory solution to the present software crisis can possibly come from a spread of software engineering practices among the engineers, coupled with further advancements to the software engineering discipline itself.

1.1.2 Program vs. software product

Programs are developed by individuals for their personal use. They are therefore, small in size and have limited functionality but software products are extremely large. In case of a program, the programmer himself is the sole user but on the other hand, in case of a software product, most users are not involved with the development. In case of a program, a single developer is involved but in case of a software product, a large number of developers are involved. For a program, the user interface may not be very important, because the programmer is the sole user. On the other hand, for a software product, user interface must be carefully designed and implemented because developers of that product and users of that product are totally different. In case of a program, very little documentation is expected, but a software product must be well documented. A program can be developed according to the programmer's individual style of development, but a software product must be developed using the accepted software engineering principles.

Check Your Progress I

1. Identify the two important techniques that software engineering uses to tackle the problem of exponential growth of problem complexity with its size.
2. State five symptoms of the present software crisis.
3. All software engineering principles are backed by either scientific basis or theoretical proof. State True or False

1.2. STRUCTURED PROGRAMMING

1.2.1 Important features of a structured program

A structured program uses three types of program constructs i.e. selection, sequence and iteration. Structured programs avoid unstructured control flows by restricting the use of GOTO statements. A structured program consists of a well partitioned set of modules. Structured programming uses single entry, single-exit program constructs such as if-then-else, do-while, etc. Thus, the structured programming principle emphasizes designing neat control structures for programs.

1.2.2 Advantages of structured programming

Structured programs are easier to read and understand. Structured programs are easier to maintain. They require less effort and time for development. They are amenable to easier debugging and usually fewer errors are made in the course of writing such programs. Evolution of software design techniques over the last 50 years.

During the 1950s, most programs were being written in assembly language. These programs were limited to about a few hundreds of lines of assembly code, i.e. were very small in size. Every programmer developed programs in his own individual style - based on his intuition. This type of programming was called Exploratory Programming.

The next significant development which occurred during early 1960s in the area computer programming was the high-level language programming. Use of high-level language programming reduced development efforts and development time significantly. Languages like FORTRAN, ALGOL, and COBOL were introduced at that time. As the size and complexity of programs kept on increasing, the exploratory programming style proved to be insufficient. Programmers found it increasingly difficult not only to write cost-effective and correct programs, but also to understand and maintain programs written by others. To cope with this problem, experienced programmers advised other programmers to pay particular attention to the design of the program's control flow structure (in late 1960s). In the late 1960s, it was found that the "GOTO" statement was the main culprit which makes control structure of a program complicated and messy. At that time most of the programmers used assembly languages extensively. They considered

use of "GOTO" statements in high-level languages were very natural because of their familiarity with JUMP statements which are very frequently used in assembly language programming. So they did not really accept that they can write programs without using GOTO statements, and considered the frequent use of GOTO statements inevitable. At this time, **Dijkstra [1968]** published his (now famous) article "GOTO Statements Considered Harmful". Expectedly, many programmers were enraged to read this article. They published several counter articles highlighting the advantages and inevitability of GOTO statements. But, soon it was conclusively proved that only three programming constructs – sequence, selection, and iteration – were sufficient to express any programming logic. This formed the basis of the structured programming methodology.

After structured programming, the next important development was data structure-oriented design. Programmers argued that for writing a good program, it is important to pay more attention to the design of data structure, of the program rather than to the design of its control structure. Data structure-oriented design techniques actually help to derive program structure from the data structure of the program. Example of a very popular data structure-oriented design technique is Jackson's Structured Programming (JSP) methodology, developed by Michael Jackson in the 1970s.

Next significant development in the late 1970s was the development of data flow-oriented design technique. Experienced programmers stated that to have a good program structure, one has to study how the data flows from input to the output of the program. Every program reads data and then processes that data to produce some output. Once the data flow structure is identified, then from there one can derive the program structure.

Object-oriented design (1980s) is the latest and very widely used technique. It has an intuitively appealing design approach in which natural objects (such as employees, pay-roll register, etc.) occurring in a problem are first identified. Relationships among objects (such as composition, reference and inheritance) are determined. Each object essentially acts as a data hiding entity.

1.2.3 Exploratory style vs. modern style of software development

An important difference is that the exploratory software development style is based on error correction while the software engineering principles are primarily based on error prevention. Inherent in the software engineering principles is the realization that it is much more cost-effective to prevent errors from occurring than to correct them as and when they are detected. Even when errors occur, software engineering principles emphasize detection of errors as close to the point where the errors are committed as possible. In the exploratory style, errors are detected only during the final product testing. In contrast, the modern practice of software development is to develop the software through several well-defined stages such as requirements specification, design, coding, testing, etc., and attempts are made to detect and fix as many errors as possible in the same phase in which they occur.

In the exploratory style, coding was considered synonymous with software development. For instance, exploratory programming style believed in developing a working system as quickly as possible and then successively modifying it until it performed satisfactorily.

In the modern software development style, coding is regarded as only a small part of the overall software development activities. There are several development activities such as design and testing which typically require much more effort than coding.

A lot of attention is being paid to requirements specification. Significant effort is now being devoted to develop a clear specification of the problem before any development activity is started. Now there is a distinct design phase where standard design techniques are employed.

Periodic reviews are being carried out during all stages of the development process. The main objective of carrying out reviews is phase containment of errors, i.e. detect and correct errors as soon as possible. Defects are usually not detected as soon as they occur, rather they are noticed much later in the life cycle. Once a defect is detected, we have to go back to the phase where it was introduced and rework those phases - possibly change the design or change the code and so on. Today, software testing has become very systematic and standard testing techniques are available. Testing activity has also become all encompassing in the

sense that test cases are being developed right from the requirements specification stage.

There is better visibility of design and code. By visibility we mean production of good quality, consistent and standard documents during every phase. In the past, very little attention was paid to producing good quality and consistent documents. In the exploratory style, the design and test activities, even if carried out (in whatever way), were not documented satisfactorily. Today, consciously good quality documents are being developed during product development. This has made fault diagnosis and maintenance smoother.

Now, projects are first thoroughly planned. Project planning normally includes preparation of various types of estimates, resource scheduling, and development of project tracking plans. Several techniques and tools for tasks such as configuration management, cost estimation, scheduling, etc. are used for effective software project management. Several metrics are being used to help in software project management and software quality assurance.

Check Your Progress II

1. State at least three differences between the exploratory style and modern styles of software development.
2. State three important advantages of structured programming.
3. Which of the following problems can be considered to be contributing to the present software crisis?
 - a. large problem size
 - b. lack of rapid progress of software engineering
 - c. lack of intelligent engineers
 - d. shortage of skilled manpower
4. Which of the following are essential program constructs (i.e. it would not be possible to develop programs for any given problem without using the construct)?
 - a. sequence
 - b. iteration
 - c. selection
 - d. jump

1.3 SUMMARY

1. Software engineering is a field of computer science, concerned with designing and writing programs for computers.
2. Software engineering appears to be among the few options available to tackle the present software crisis. To explain the present software crisis in simple words, it is considered the following that are being faced. The expenses that organizations all around the world are incurring on software purchases compared to those on hardware purchases have been showing a worrying trend over the years.
3. There are many factors that have contributed to the making of the present software crisis. Those factors are larger problem sizes, lack of adequate training in software engineering, increasing skill shortage, and low productivity improvements.
4. Programs are developed by individuals for their personal use. They are therefore, small in size and have limited functionality but software products are extremely large.
5. In case of a program, the programmer himself is the sole user but on the other hand, in case of a software product, a large number of users who are not involved with the development are attached. In case of a program, a single developer is involved but in case of a software product, a large number of developers are involved.
6. In case of a program, very little documentation is expected but a software product must be well documented. A program can be developed according to the programmer's individual style of development but a software product must be developed using software engineering principles.
7. A structured program uses three type of program constructs i.e. selection, sequence and iteration.
8. The structured programming principle emphasizes designing neat control structures for programs.
9. Structured programs are easier to maintain and they require less effort and time for development.
10. The exploratory software development style is based on error correction while the software engineering principles are primarily based on error prevention.

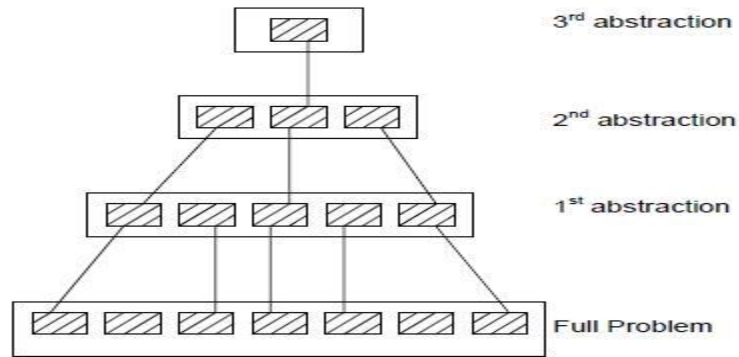
11. As the size and complexity of programs kept on increasing, the exploratory programming style proved to be insufficient.
12. Assembly language programs are limited to about a few hundreds of lines of assembly code, i.e. are very small in size.
13. But use of high-level programming language reduces development efforts and development time significantly.
14. Control flow-oriented design deals with carefully designing the program's control structure.
15. A program's control structure refers to the sequence, in which the program's instructions are executed

1.4 MODEL QUESTIONS

1. Identify the problem one would face, if he tries to develop a large software product without using software engineering principles.\
2. State four factors that have contributed to the making of the present software crisis.
3. Suggest at least two possible solutions to the present software crisis.
4. Identify at least four basic characteristics that differentiate a simple program from a software product.
5. Identify two important features of that a program must satisfy to be called as a structured program.
6. Explain exploratory program development style.
7. Show at least three important drawbacks of the exploratory programming style.
8. Identify at least two advantages of using high-level languages over assembly languages.
9. State at least two basic differences between control flow-oriented and data flow-oriented design techniques.
10. State at least five advantages of object-oriented design techniques.

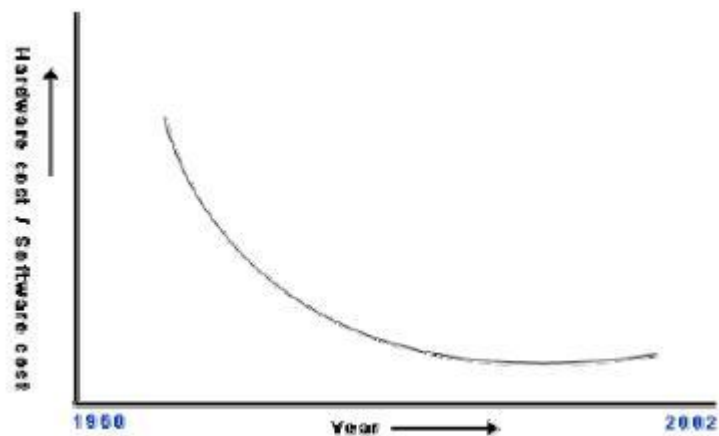
Answers to Check Your Progress I

1. Software engineering principles use two important techniques to reduce problem complexity: abstraction and decomposition.



The principle of abstraction implies that a problem can be simplified by omitting irrelevant details. Once simpler problem is solved then the omitted details can be taken into consideration to solve the next lower level abstraction. In this technique any random decomposition of a problem into smaller parts will not help. The problem has to be decomposed such that each component of the decomposed problem can be solved in solution and then the solution of the different components can be combined to get the full solution.

2. Software engineering appears to be among the few options available to tackle the present software crisis. To explain the present software crisis in simple words, it is considered the following that are being faced. The expenses that organizations all around the world are incurring on software purchases compared to those on hardware purchases have been showing a worrying trend over the years.



Organizations are spending larger and larger portions of their budget on software. Not only are the software products turning out to be more expensive than hardware, but they also present a host of other problems to the customers: software products are difficult to alter, debug, and enhance; use resources non-optimally; often fail to meet the user requirements; are far from being reliable; frequently crash; and are often delivered late. Among these, the trend of increasing software costs is probably the most important symptom of the present software crisis.

3. False

Answers to Check Your Progress II

1. An important difference is that the exploratory software development style is based on error correction while the software engineering principles are primarily based on error prevention. Inherent in the software engineering principles is the realization that it is much more.
2. Structured programs are easier to read and understand. Structured programs are easier to maintain. They require less effort and time for development. They are amenable to easier debugging and usually fewer errors are made in the course of writing such programs.
3. a, b and d
4. a, b and c

UNIT II: SOFTWARE LIFE CYCLE MODEL

2.0 LEARNING OBJECTIVES

At the end of this lesson the student will be able to:

- Explain what is a life cycle model.
- Explain what problems would occur if no life cycle model is followed.
- Identify the different software life cycle models.
- Identify the different phases of the classical waterfall model.
- Identify the activities undertaken in each phase.
- Identify the shortcomings of the classical waterfall model.
- Identify the phase-entry and phase-exit criteria of each phase.
- Explain what a prototype is.
- Explain why and when a prototype needs to be developed during software development.
- Identify the situations in which one would prefer to build a prototype.
- State the activities carried out during each phase of a spiral model.
- Identify circumstances under which spiral model should be used for software development.
- Tailor a development process to a specific project

2.1 BASICS OF SOFTWARE LIFE CYCLE AND WATERFALL MODEL

2.1.1 Life cycle model

A software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle. A life cycle model represents all the activities required to make a software product transit through its life cycle phases. It also captures the order in which these activities are to be undertaken. In other words, a life cycle model maps the different activities performed on a software product from its inception to retirement. Different life cycle models may map the basic development activities to phases in different

ways. Thus, no matter which life cycle model is followed, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different life cycle models. During any life cycle phase, more than one activity may also be carried out. For example, the design phase might consist of the structured analysis activity followed by the structured design activity.

2.1.2 The need for a software life cycle model

The development team must identify a suitable life cycle model for the particular project and then adhere to it. Without using of a particular life cycle model the development of a software product would not be in a systematic and disciplined manner. When a software product is being developed by a team there must be a clear understanding among team members about when and what to do. Otherwise it would lead to chaos and project failure. This problem can be illustrated by using an example. Suppose a software development problem is divided into several parts and the parts are assigned to the team members. From then on, suppose the team members are allowed the freedom to develop the parts assigned to them in whatever way they like. It is possible that one member might start writing the code for his part, another might decide to prepare the test documents first, and some other engineer might begin with the design phase of the parts assigned to him. This would be one of the perfect recipes for project failure.

A software life cycle model defines entry and exit criteria for every phase. A phase can start only if its phase-entry criteria have been satisfied. So without software life cycle model the entry and exit criteria for a phase cannot be recognized. Without software life cycle models (such as classical waterfall model, iterative waterfall model, prototyping model, evolutionary model, spiral model etc.) it becomes difficult for software project managers to monitor the progress of the project.

2.1.3 Different software life cycle models

Many life-cycle models have been proposed so far. Each of them has some advantages as well as some disadvantages. A few important and commonly used life cycle models are as follows:

- Classical Waterfall Model
- Iterative Waterfall Model
- Prototyping Model
- Evolutionary Model
- Spiral Model

2.1.4 Different phases of the classical waterfall model

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, it is not a practical model in the sense that it cannot be used in actual software development projects. Thus, this model can be considered to be a *theoretical way of developing software*. But all other life cycle models are essentially derived from the classical waterfall model. So, in order to be able to appreciate other life cycle models it is necessary to learn the classical waterfall model. Classical waterfall model divides the life cycle into the following phases as shown in fig.2.1:

Feasibility Study

- Requirements Analysis and Specification
- Design
- Coding and Unit Testing
- Integration and System Testing
- Maintenance

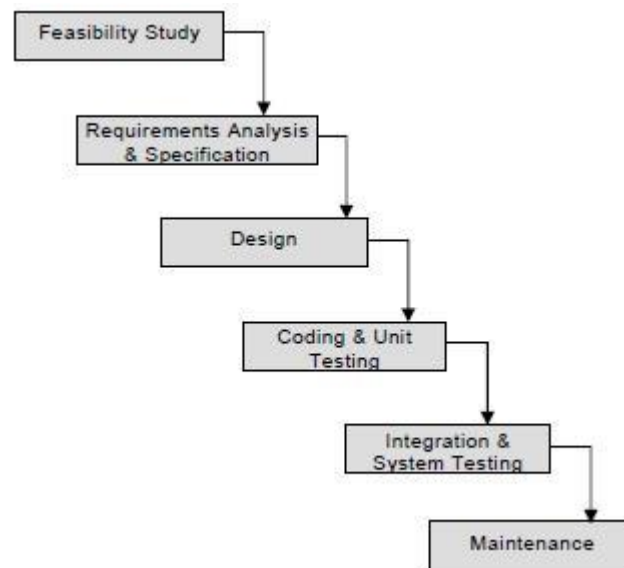


Fig 2.1: Classical Waterfall Model

2.1.4.1 Activities in each phase of the life cycle

Activities undertaken during feasibility study

The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product. At first project managers or team leaders try to have a rough understanding of what is required to be done by visiting the client side. They study different input data to the system and output data to be produced by the system. They study what kind of processing is needed to be done on these data and they look at the various constraints on the behavior of the system. After they have an overall understanding of the problem, they investigate the different solutions that are possible. Then they examine each of the solutions in terms of what kind of resources required, what would be the cost of development and what would be the development time for each solution. Based on this analysis they pick the best solution and determine whether the solution is feasible financially and technically. They check whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in the area of development. The following is an example of a feasibility study undertaken by an organization. It is intended to give you a feel of the activities and issues involved in the feasibility study phase of a typical software project.

4.1.4.2 Case Study

A mining company named Galaxy Mining Company Ltd. (GMC) has mines located at various places in India. It has about fifty different mine sites spread across eight states. The company employs a large number of mines at each mine site. Mining being a risky profession, the company intends to operate a special provident fund, which would exist in addition to the standard provident fund that the miners already enjoy. The main objective of having the special provident fund (SPF) would be quickly distribute some compensation before the standard provident amount is paid. According to this scheme, each mine site would deduct SPF installments from each miner every month and deposit the same with the CSPFC (Central Special Provident Fund Commissioner). The CSPFC will maintain all details regarding the SPF installments collected from the miners. GMC employed a reputed software vendor Adventure Software Inc. to undertake

the task of developing the software for automating the maintenance of SPF records of all employees. GMC realized that besides saving manpower on bookkeeping work, the software would help in speedy settlement of claim cases. GMC indicated that the amount it can afford for this software to be developed and installed is Rs. 1 million.

Adventure Software Inc. deputed their project manager to carry out the feasibility study. The project manager discussed the matter with the top managers of GMC to get an overview of the project. He also discussed the issues involved with the several field PF officers at various mine sites to determine the exact details of the project. The project manager identified two broad approaches to solve the problem. One was to have a central database which could be accessed and updated via a satellite connection to various mine sites. The other approach was to have local databases at each mine site and to update the central database periodically through a dial-up connection. These periodic updates could be done on a daily or hourly basis depending on the delay acceptable to GMC in invoking various functions of the software. The project manager found that the second approach was very affordable and more fault-tolerant as the local mine sites could still operate even when the communication link to the central database temporarily failed. The project manager quickly analyzed the database functionalities required, the user-interface issues, and the software handling communication with the mine sites. He arrived at a cost to develop from the analysis. He found that the solution involving maintenance of local databases at the mine sites and periodic updating of a central database was financially and technically feasible. The project manager discussed his solution with the GMC management and found that the solution was acceptable to them as well.

Activities undertaken during requirements analysis and specification

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

- Requirements gathering and analysis, and
- Requirements specification

The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done

to clearly understand the customer requirements so that incompleteness and inconsistencies are removed.

The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements. The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system. Therefore, it is necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document. The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document. The important components of this document are functional requirements, the non-functional requirements, and the goals of implementation.

Activities undertaken during design

The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the traditional design approach and the object-oriented design approach.

Traditional design approach

Traditional design consists of two different activities; first a structured analysis of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.

Object-oriented design approach

In this technique, various objects that occur in the problem domain and the solution domain are first identified, and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design.

Activities undertaken during coding and unit testing

The purpose of the coding and unit testing phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested. During this phase, each module is unit tested to determine the correct working of all the individual modules. It involves testing each module in isolation as this is the most efficient way to debug the errors identified at this stage.

Activities undertaken during integration and system testing

Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the modules are integrated in a planned manner. The different modules making up a software product are almost never integrated in one shot. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to its requirements laid out in the SRS document. System testing usually consists of three different kinds of testing activities:

- α – testing: It is the system testing performed by the development team.
- β – Testing: It is the system testing performed by a friendly set of customers.
- Acceptance testing: It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

System testing is normally carried out in a planned manner according to the system test plan document. The system test plan identifies all testing-related activities that must be performed, specifies the schedule of testing, and allocates resources. It also lists all the test cases and the expected outputs for each test case.

Activities undertaken during maintenance

Maintenance of a typical software product requires much more than the effort necessary to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratio. Maintenance involves performing any one or more of the following three kinds of activities:

- Correcting errors that were not discovered during the product development phase. This is called corrective maintenance.
- Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements. This is called perfective maintenance.
- Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system. This is called adaptive maintenance.

Shortcomings of the classical waterfall model

The classical waterfall model is an idealistic one since it assumes that no development error is ever committed by the engineers during any of the life cycle phases. However, in practical development environments, the engineers do commit a large number of errors in almost every phase of the life cycle. The source of the defects can be many: oversight, wrong assumptions, use of inappropriate technology, communication gap among the project engineers, etc. These defects usually get detected much later in the life cycle. For example, a design defect might go unnoticed till we reach the coding or testing phase. Once a defect is detected, the engineers need to go back to the phase where the defect had occurred and redo some of the work done during that phase and the subsequent phases to correct the defect and its effect on the later phases. Therefore, in any practical software development work, it is not possible to strictly follow the classical waterfall model.

Phase-entry and phase-exit criteria of each phase

At the starting of the feasibility study, project managers or team leaders try to understand what is the actual problem by visiting the client side. At the end of that phase they pick the best solution and determine whether the solution is feasible financially and technically. At the starting of requirements analysis and specification phase the required data is collected. After that requirement specification is carried out. Finally, SRS document is produced. At the starting of design phase, context diagram and different levels of DFDs are produced according to the SRS document. At the end of this phase module structure (structure chart) is produced.

During the coding phase each module (independently compilation unit) of the design is coded. Then each module is tested independently as a stand-alone unit and debugged separately. After this each module is documented individually. The end product of the implementation phase is a set of program modules that have been tested individually but not tested together.

After the implementation phase, different modules which have been tested individually are integrated in a planned manner. After all the modules have been successfully integrated and tested, system testing is carried out.

Software maintenance denotes any changes made to a software product after it has been delivered to the customer. Maintenance is inevitable for almost any kind of product. However, most products need maintenance due to the wear and tear caused by use.

Check Your Progress I

1. In a classical waterfall model, which phase precedes the design phase?
 - a. Coding and unit testing
 - b. Maintenance
 - c. Requirements analysis and specification
 - d. Feasibility study
2. Among development phases of software life cycle, which phase typically consumes the maximum effort?
 - a. Requirements analysis and specification
 - b. Design
 - c. Coding
 - d. Testing

3. Among all the phases of software life cycle, which phase consumes the maximum effort?
 - a. Design
 - b. Maintenance
 - c. Testing
 - d. Coding
4. Which phase is the last development phase of a classical waterfall software life cycle?
 - a. Design
 - b. Maintenance
 - c. Testing
 - d. Coding
5. Which development phase in classical waterfall life cycle immediately follows coding phase?
 - a. Design
 - b. Maintenance
 - c. Testing
 - d. Requirement analysis and specification

2.2. PROTOTYPING AND SPIRAL LIFE CYCLE MODELS

2.2.1 Prototype

A prototype is a toy implementation of the system. A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software. A prototype is usually built using several shortcuts. The shortcuts might involve using inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up instead of performing the actual computations. A prototype usually turns out to be a very crude version of the actual system.

2.2.1.1 Need for a prototype in software development

There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer's needs:

- How the screens might look like

- How the user interface would behave
- How the system would produce outputs

This is something similar to what the architectural designers of a building do; they show a prototype of the building to their customer. The customer can evaluate whether he likes it or not and the changes that he would need in the actual product. A similar thing happens in the case of a software product and its prototyping model.

Another reason for developing a prototype is that it is impossible to get the perfect product in the first attempt. Many researchers and engineers advocate that if you want to develop a good product you must plan to throw away the first version. The experience gained in developing the prototype can be used to develop the final product.

A prototyping model can be used when technical solutions are unclear to the development team. A developed prototype can help engineers to critically examine the technical issues associated with the product development. Often, major design decisions depend on issues like the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype may be the best or the only way to resolve the technical issues.

2.2.1.2 Examples for prototype model

A prototype of the actual product is preferred in situations such as:

- User requirements are not complete
- Technical issues are not clear

Let's see an example for each of the above category.

Example 1: User requirements are not complete

In any application software like billing in a retail shop, accounting in a firm, etc the users of the software are not clear about the different functionalities required. Once they are provided with the prototype implementation, they can try to use it and find out the missing functionalities.

Example 2: Technical issues are not clear

Suppose a project involves writing a compiler and the development team has never written a compiler. In such a case, the team can consider a simple language, try to build a compiler in order to check the issues that arise in the process and

resolve them. After successfully building a small compiler (prototype), they would extend it to one that supports a complete language.

2.2.2 Spiral model

The Spiral model of software development is shown in fig. 2.2. The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study. The next loop with requirements specification, the next one with design, and so on. Each phase in this model is split into four sectors (or quadrants) as shown in fig. 2.2. The following activities are carried out during each phase of a spiral model.

- **First quadrant (Objective Setting)**
 - During the first quadrant, it is needed to identify the objectives of the phase. Examine the risks associated with these objectives.
- **Second Quadrant (Risk Assessment and Reduction)**
 - A detailed analysis is carried out for each identified project risk. Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

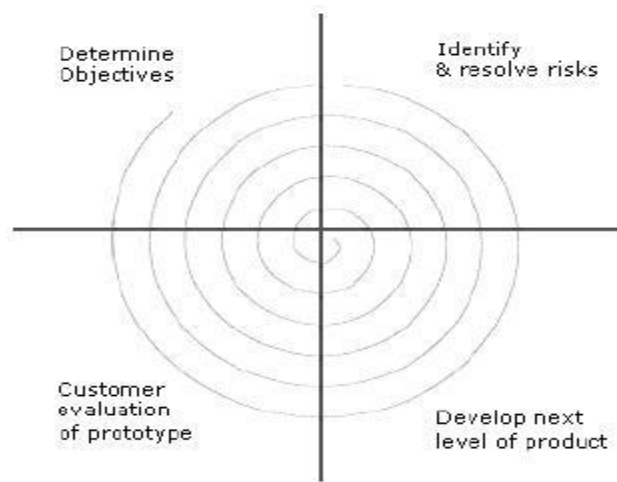


Fig. 2.2: Spiral Model

- **Third Quadrant (Development and Validation)**
 - Develop and validate the next level of the product after resolving identified risks.
- **Fourth Quadrant (Review and Planning)**
 - Review the results achieved so far with the customer and plan the next iteration around the spiral.

- Progressively more complete version of the software gets built with each iteration around the spiral.

2.2.2.1 Circumstances to use spiral model

The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

2.2.3 Comparison of different life-cycle models

The classical waterfall model can be considered as the basic model and all other life cycle models as embellishments of this model. However, the classical waterfall model can not be used in practical development projects, since this model supports no mechanism to handle the errors committed during any of the phases.

This problem is overcome in the iterative waterfall model. The iterative waterfall model is probably the most widely used software development model evolved so far. This model is simple to understand and use. However, this model is suitable only for well-understood problems; it is not suitable for very large projects and for projects that are subject to many risks.

The prototyping model is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood. This model is especially popular for development of the user-interface part of the projects.

The evolutionary approach is suitable for large problems which can be decomposed into a set of modules for incremental development and delivery. This model is also widely used for object-oriented development projects. Of course, this model can only be used if the incremental delivery of the system is acceptable to the customer.

The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are

prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

The different software life cycle models can be compared from the viewpoint of the customer. Initially, customer confidence in the development team is usually high irrespective of the development model followed. During the lengthy development process, customer confidence normally drops off, as no working product is immediately visible. Developers answer customer queries using technical slang, and delays are announced. This gives rise to customer resentment. On the other hand, an evolutionary approach lets the customer experiment with a working product much earlier than the monolithic approaches. Another important advantage of the incremental model is that it reduces the customer's trauma of getting used to an entirely new system. The gradual introduction of the product via incremental phases provides time to the customer to adjust to the new product. Also, from the customer's financial viewpoint, incremental development does not require a large upfront capital outlay. The customer can order the incremental versions as and when he can afford them.

Check your Progress II

1. Evolutionary life cycle model is ideally suited for development of very small software products typically requiring a few months of development effort. (True/False)
2. Which of the following life cycle model can be chosen if the development team has less experience on similar projects?
 - a. Spiral
 - b. Waterfall
 - c. RAD
 - d. Iterative Enhancement Model
3. SDLC stands for
 - a. Software Development Life Cycle
 - b. System Development Life cycle
 - c. Software Design Life Cycle
 - d. System Design Life Cycle

4. Prototyping life cycle model is the most suitable one for undertaking a software development project susceptible to schedule slippage.(True/False)
5. Spiral life cycle model is not suitable for products that are vulnerable to large number of risks. (True/False)

2.3 SUMMARY

1. The development team must identify a suitable life cycle model for the particular project and then adhere to it.
2. Without using of a particular life cycle model the development of a software product would not be in a systematic and disciplined manner.
3. When a software product is being developed by a team there must be a clear understanding among team members about when and what to do. Otherwise it would lead to chaos and project failure.
4. A software life cycle model defines entry and exit criteria for every phase.
5. A phase can start only if its phase-entry criteria have been satisfied. So, without software life cycle model the entry and exit criteria for a phase cannot be recognized.
6. Without software life cycle models (such as classical waterfall model, iterative waterfall model, prototyping model, evolutionary model, spiral model etc.) it becomes difficult for software project managers to monitor the progress of the project.
7. For performing requirements analysis activity system analyst collects all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions.
8. The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system. Therefore, a system analyst identifies all ambiguities and contradictions in the requirements and resolves them through further discussions with the customer.
9. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the system analyst starts requirements specification activity.

10. Traditional design consists of two different activities; first a structured analysis of the requirements specification is carried out where the detailed structure of the problem is examined.
11. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.
12. During unit testing, each module is unit tested to determine the correct working of all the individual modules. It involves testing each module in isolation as this is the most efficient way to debug the errors identified at this stage. So, it is always a good idea to test a module in isolation from other modules.
13. Integration of different modules is undertaken once they have been coded and unit tested.
14. During the integration and system testing phase, the modules are integrated in a planned manner.
15. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out.
16. A prototype is a toy implementation of the system. A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software. A prototype is usually built using several shortcuts.
17. The spiral model is called a meta model since it encompasses all other life cycle models.
18. Risk handling is inherently built into this model.
19. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks.

2.4 MODEL QUESTIONS

1. Identify the definite stages through which a software product undergoes during its lifetime.
2. Explain the problems that might be faced by an organization if it does not follow any software life cycle model.

3. Identify six different phases of a classical waterfall model.
4. Identify two basic roles of a system analyst.
5. Differentiate between structured analysis and structured design.
6. Identify at least three activities undertaken in an object-oriented software design approach.
7. State why it is a good idea to test a module in isolation from other modules.
8. Identify why different modules making up a software product are almost never integrated in one shot.
9. Mention at least two reasons as to why classical waterfall model can be considered impractical and cannot be used in real projects.
10. Explain what a software prototype is?
11. Identify three reasons for the necessity of developing a prototype during software development.
12. Identify when a prototype needs to develop.
13. Identify at least two activities carried out during each phase of a spiral model.
14. Write down the two advantages of using spiral model.

Answers to Check Your Progress I

1. c
2. d
3. b
4. c
5. c

Answers to Check your Progress II

1. False
2. a
3. a
4. False
5. False

UNIT III: REQUIREMENTS ANALYSIS AND SPECIFICATION

3.0 LEARNING OBJECTIVES

At the end of this lesson the student will be able to:

- Explain the role of a system analyst.
- Identify the important parts of SRS document.
- Identify the functional requirements from any given problem description.
- Document the functional requirements from any given problem description.
- Identify the important properties of a good SRS document.
- Identify the important problems that an organization would face if it does not develop an SRS document.
- Identify non-functional requirements from any given problem description.
- Identify the problems that an unstructured specification would create during software development.
- Represent complex conditions in the form of a decision tree.
- Represent complex conditions in the form of decision table.
- Explain what a formal technique is.
- Explain what a formal specification language is.
- Differentiate between model-oriented and property-oriented approaches in the context of requirements specification.
- Explain the operational semantics of a formal method.
- Identify the merits of formal requirements specification.
- Identify the limitations of formal requirements specification.
- Develop axiomatic specification of simple problems.
- Explain algebraic specification and its use.
- Explain how algebraic specifications are represented.
- Develop algebraic specification of simple problems.

- Identify the basic properties that a good algebraic specification should satisfy.
- State the properties of a structured specification.
- State the advantages and disadvantages of algebraic specifications.
- State the features of an executable specification language (4GL) with suitable examples.

3.1 BASIC CONCEPTS IN REQUIREMENTS ANALYSIS AND SPECIFICATION

3.1.1 Role of a system analyst

The analyst starts requirements gathering and analysis activity by collecting all information from the customer which could be used to develop the requirements of the system. He then analyzes the collected information to obtain a clear and thorough understanding of the product to be developed, with a view to removing all ambiguities and inconsistencies from the initial customer perception of the problem. The following basic questions pertaining to the project should be clearly understood by the analyst in order to obtain a good grasp of the problem:

- What is the problem?
- Why is it important to solve the problem?
- What are the possible solutions to the problem?
- What exactly are the data input to the system and what exactly are the data output by the system?
- What are the likely complexities that might arise while solving the problem?
- If there are external software or hardware with which the developed software has to interface, then what exactly would the data interchange formats with the external system be?

After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various requirements problems. The most important requirements problems that the analyst has to identify and eliminate are the problems of anomalies, inconsistencies, and incompleteness. When the analyst

detects any inconsistencies, anomalies or incompleteness in the gathered requirements, he resolves them by carrying out further discussions with the end-users and the customers.

3.1.2 Parts of a SRS document

The important parts of SRS document are:

- Functional requirements of the system
- Non-functional requirements of the system, and
- Goals of implementation

Functional requirements

- The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high-level functions $\{f_i\}$. The functional view of the system is shown in fig. 3.1. Each function f_i of the system can be considered as a transformation of a set of input data (i_i) to the corresponding set of output data (o_i). The user can get some meaningful piece of work done using a high-level function.

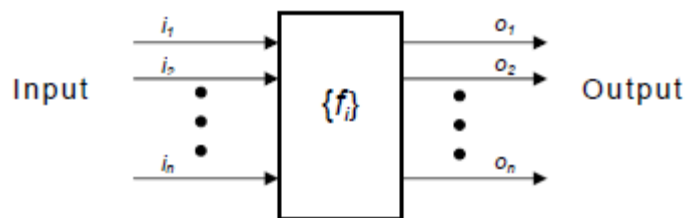


Fig. 3.1: View of a system performing a set of functions

Nonfunctional requirements

- Nonfunctional requirements deal with the characteristics of the system which can not be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.
- Nonfunctional requirements may include:
 - # Reliability issues,
 - # Accuracy of results,
 - # Human - computer interface issues,
 - # Constraints on the system implementation etc.

Goals of implementation

The goals of implementation part documents some general suggestions regarding development. These suggestions guide trade-off among design goals. The goals of implementation section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.

3.1.3 Identifying functional requirements from a problem description

The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem. Each high-level requirement characterizes a way of system usage by some user to perform some meaningful piece of work. There can be many types of users of a system and their requirements from the system may be very different. So, it is often useful to identify the different types of users who might use the system and then try to identify the requirements from each user's perspective.

Here we list all functions $\{f_i\}$ that the system performs. Each function f_i as shown in fig.3.2 is considered as a transformation of a set of input data to some corresponding output data.

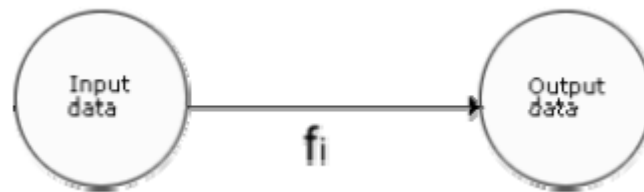


Fig. 3.2: Function f_i

Example: - Consider the case of the library system, where - **F1:** Search Book function (fig. 3.3) **Input:** an author's name **Output:** details of the author's books and the location of these books in the library

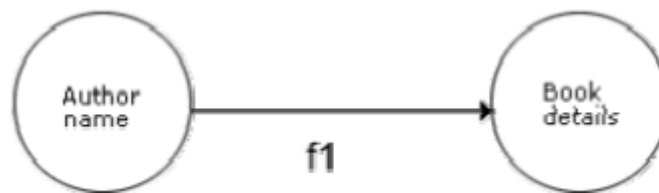


Fig. 3.3: Book Function

So the function Search Book (F1) takes the author's name and transforms it into book details.

Functional requirements actually describe a set of high-level requirements, where each high-level requirement takes some data from the user and provides some data to the user as an output. Also, each high-level requirement might consist of several other functions.

Documenting functional requirements

For documenting the functional requirements, we need to specify the set of functionalities supported by the system. A function can be specified by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data. Let us first try to document the withdraw-cash function of an ATM (Automated Teller Machine) system. The withdraw-cash is a high-level requirement. It has several sub-requirements corresponding to the different user interactions. These different interaction sequences capture the different scenarios.

Example: - Withdraw Cash from ATM

R1: withdraw cash

Description: The withdraw cash function first determines the type of account that the user has and the account number from which the user wishes to withdraw cash. It checks the balance to determine whether the requested amount is available in the account. If enough balance is available, it outputs the required cash, otherwise it generates an error message.

R1.1 select withdraw amount option

Input: “withdraw amount” option

Output: user prompted to enter the account type

R1.2: select account type

Input: user option

Output: prompt to enter amount

R1.3: get required amount

Input: amount to be withdrawn in integer values greater than 100 and less than 10,000 in multiples of 100.

Output: The requested cash and printed transaction statement.

Processing: the amount is debited from the user's account if sufficient balance is available, otherwise an error message displayed.

Properties of a good SRS document

The important properties of a good SRS document are the following:

- **Concise.** The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase error possibilities.
- **Structured.** It should be well-structured. A well-structured document is easy to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the customer requirements. Often, the customer requirements evolve over a period of time. Therefore, in order to make the modifications to the SRS document easy, it is important to make the document well-structured.
- **Black-box view.** It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the external behavior of the system and not discuss the implementation issues. The SRS document should view the system to be developed as black box, and should specify the externally visible behavior of the system. For this reason, the SRS document is also called the black-box specification of a system.
- **Conceptual integrity.** It should show conceptual integrity so that the reader can easily understand it.
- **Response to undesired events.** It should characterize acceptable responses to undesired events. These are called system response to exceptional conditions.
- **Verifiable.** All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to determine whether or not requirements have been met in an implementation.

3.1.4 Problems without a SRS document

The important problems that an organization would face if it does not develop an SRS document are as follows:

- Without developing the SRS document, the system would not be implemented according to customer needs.

- Software developers would not know whether what they are developing is what exactly required by the customer.
- Without SRS document, it will be very much difficult for the maintenance engineers to understand the functionality of the system.
- It will be very much difficult for user document writers to write the users' manuals properly without understanding the SRS document.

3.1.5 Identifying non-functional requirements

Nonfunctional requirements are the characteristics of the system which can not be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc. Nonfunctional requirements may include:

- Reliability issues,
- Performance issues,
- Human - computer interface issues,
- interface with other external systems,
- Security and maintainability of the system, etc.

Problems with an unstructured specification

- It would be very much difficult to understand that document.
- It would be very much difficult to modify that document.
- Conceptual integrity in that document would not be shown.
- The SRS document might be unambiguous and inconsistent.

3.1.6 Decision tree

A decision tree gives a graphic view of the processing logic involved in decision making and the corresponding actions taken. The edges of a decision tree represent conditions and the leaf nodes represent the actions to be performed depending on the outcome of testing the condition.

Example

Consider Library Membership Automation Software (LMS) where it should support the following three options:

- **New member**
- **Renewal**
- **Cancel membership**

New member option

Decision: When the 'new member' option is selected, the software asks details about the member like the member's name, address, phone number etc.

Action: If proper information is entered then a membership record for the member is created and a bill is printed for the annual membership charge plus the security deposit payable.

Renewal option-

Decision: If the 'renewal' option is chosen, the LMS asks for the member's name and his membership number to check whether he is a valid member or not.

Action: If the membership is valid then membership expiry date is updated and the annual membership bill is printed, otherwise an error message is displayed.

Cancel membership option

Decision: If the 'cancel membership' option is selected, then the software asks for member's name and his membership number.

Action: The membership is cancelled, a cheque for the balance amount due to the member is printed and finally the membership record is deleted from the database.

Decision tree representation of the above example - The following tree (fig. 3.4) shows the graphical representation of the above example. After getting information from the user, the system makes a decision and then performs the corresponding actions.

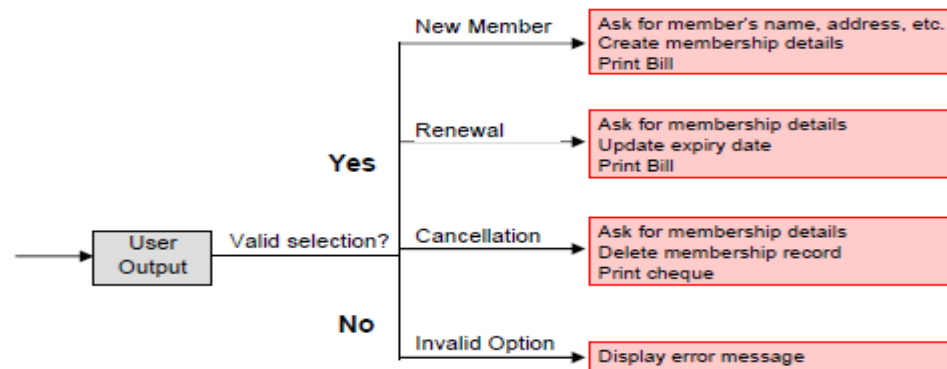


Fig. 3.4: Decision tree for LMS

3.1.7 Decision table

A decision table is used to represent the complex processing logic in a tabular or a matrix form. The upper rows of the table specify the variables or conditions to be evaluated. The lower rows of the table specify the actions to be taken when the

corresponding conditions are satisfied. A column in a table is called a *rule*. A rule implies that if a condition is true, then the corresponding action is to be executed.

Example: - Consider the previously discussed LMS example. The following decision table (fig. 3.5) shows how to represent the LMS problem in a tabular form. Here the table is divided into two parts, the upper part shows the conditions and the lower part shows what actions are taken. Each column of the table is a rule.

Conditions				
Valid selection	No	Yes	Yes	Yes
New member	-	Yes	No	No
Renewal	-	No	Yes	No
Cancellation	-	No	No	Yes
Actions				
Display error message	x	-	-	-
Ask member's details	-	x	-	-
Build customer record	-	x	-	-
Generate bill	-	x	x	-
Ask member's name & membership number	-	-	x	x
Update expiry date	-	-	x	-
Print cheque	-	-	-	x
Delete record	-	-	-	x

Fig. 3.5: Decision table for LMS

From the above table you can easily understand that, if the valid selection condition is false then the action taken for this condition is 'display error message'. Similarly, the actions taken for other conditions can be inferred from the table.

Check Your Progress I

1. An SRS document normally contains
 - a. Functional requirements of the system
 - b. Module structure
 - c. Configuration management plan
 - d. Non-functional requirements of the system
 - e. Constraints on the system
2. The structured specification technique that is used to reduce the effort in writing specification is
 - a. Incremental specification
 - b. Specification instantiation
 - c. Both of the above

- d. None of the above
3. Examples of executable specifications are
- a. Third generation languages
 - b. Fourth generation languages
 - c. Second-generation languages
 - d. First generation languages

3.2. FORMAL REQUIREMENTS SPECIFICATION

3.2.1 Formal technique

A formal technique is a mathematical method to specify a hardware and/or software system, verify whether a specification is realizable, verify that an implementation satisfies its specification, prove properties of a system without necessarily running the system, etc. The mathematical basis of a formal method is provided by the specification language.

3.2.1.1 Formal specification language

A formal specification language consists of two sets syn and sem , and a relation sat between them. The set syn is called the syntactic domain, the set sem is called the semantic domain, and the relation sat is called the satisfaction relation. For a given specification syn , and model of the system sem , if $sat(syn, sem)$, as shown in fig. 3.6, then syn is said to be the specification of sem , and sem is said to be the specific and of syn .



Fig. 3.6: $sat(syn, sem)$

Syntactic Domains

The syntactic domain of a formal specification language consists of an alphabet of symbols and set of formation rules to construct well-formed formulas from the alphabet. The well-formed formulas are used to specify a system.

Semantic Domains

Formal techniques can have considerably different semantic domains. Abstract data type specification languages are used to specify algebras, theories, and programs. Programming languages are used to specify functions from input to output values. Concurrent and distributed system specification languages are used to specify state sequences, event sequences, state-transition sequences, synchronization trees, partial orders, state machines, etc.

Satisfaction Relation

Given the model of a system, it is important to determine whether an element of the semantic domain satisfies the specifications. This satisfaction is determined by using a homomorphism known as semantic abstraction function. The semantic abstraction function maps the elements of the semantic domain into equivalent classes. There can be different specifications describing different aspects of a system model, possibly using different specification languages. Some of these specifications describe the system's behavior and the others describe the system's structure. Consequently, two broad classes of semantic abstraction functions are defined: those that preserve a system's behavior and those that preserve a system's structure.

3.2.2 Model-oriented vs. property-oriented approaches

Formal methods are usually classified into two broad categories – model – oriented and property – oriented approaches. In a model-oriented style, one defines a system's behavior directly by constructing a model of the system in terms of mathematical structures such as tuples, relations, functions, sets, sequences, etc.

In the property-oriented style, the system's behavior is defined indirectly by stating its properties, usually in the form of a set of axioms that the system must satisfy.

Example

Let us consider a simple producer/consumer example. In a property-oriented style, it is probably started by listing the properties of the system like: the consumer can start consuming only after the producer has produced an item, the producer starts to produce an item only after the consumer has consumed the last item, etc. A good example of a producer-consumer problem is CPU-Printer coordination. After processing of data, CPU outputs characters to the buffer for printing. Printer, on

the other hand, reads characters from the buffer and prints them. The CPU is constrained by the capacity of the buffer, whereas the printer is constrained by an empty buffer. Examples of property-oriented specification styles are axiomatic specification and algebraic specification.

In a model-oriented approach, we start by defining the basic operations, p (produce) and c (consume). Then we can state that $S1 + p \rightarrow S$, $S + c \rightarrow S1$. Thus the model-oriented approaches essentially specify a program by writing another, presumably simpler program. Examples of popular model-oriented specification techniques are Z, CSP, CCS, etc.

Model-oriented approaches are more suited to use in later phases of life cycle because here even minor changes to a specification may lead to drastic changes to the entire specification. They do not support logical conjunctions (AND) and disjunctions (OR).

Property-oriented approaches are suitable for requirements specification because they can be easily changed. They specify a system as a conjunction of axioms and you can easily replace one axiom with another one.

Operational semantics

Informally, the operational semantics of a formal method is the way computations are represented. There are different types of operational semantics according to what is meant by a single run of the system and how the runs are grouped together to describe the behavior of the system. Some commonly used operational semantics are as follows:

Linear Semantics

In this approach, a run of a system is described by a sequence (possibly infinite) of events or states. The concurrent activities of the system are represented by non-deterministic interleavings of the automatic actions. For example, a concurrent activity $a \parallel b$ is represented by the set of sequential activities $a;b$ and $b;a$. This is simple but rather unnatural representation of concurrency. The behavior of a system in this model consists of the set of all its runs. To make this model realistic, usually justice and fairness restrictions are imposed on computations to exclude the unwanted interleavings.

3.2.3 Branching Semantics

In this approach, the behavior of a system is represented by a directed graph as shown in the fig. 3.7. The nodes of the graph represent the possible states in the evolution of a system. The descendants of each node of the graph represent the states which can be generated by any of the atomic actions enabled at that state. An example involving the transactions in an ATM is shown in fig. 3.7. Although this semantic model distinguishes the branching points in a computation, still it represents concurrency by interleaving.

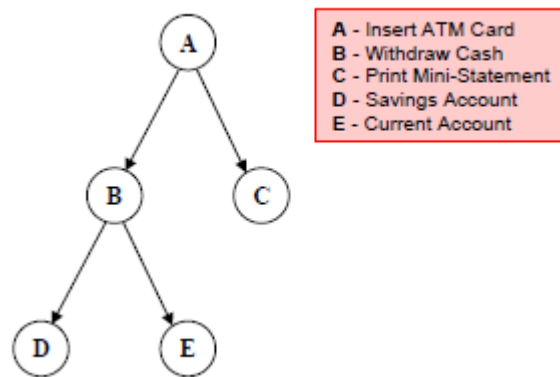


Fig. 3.7: Branching semantics

Maximally parallel semantics

In this approach, all the concurrent actions enabled at any state are assumed to be taken together. This is again not a natural model of concurrency since it implicitly assumes the availability of all the required computational resources.

Partial order semantics

Under this view, the semantics ascribed to a system is a structure of states satisfying a partial order relation among the states (events). The partial order represents a precedence ordering among events, and constraints some events to occur only after some other events have occurred; while the occurrence of other events (called concurrent events) is considered to be incomparable. This fact identifies concurrency as a phenomenon not translatable to any interleaved representation.

For example, figure (fig. 3.8) shows the semantics implied by a simplified beverage selling machine. From the figure, we can infer that beverage is dispensed only if an inserted coin is accepted by the machine (precedence). Similarly, preparation of ingredients and milk are done simultaneously (concurrency). Hence,

node Ingredient can be compared with node Brew, but neither can it be compared with node Hot/Cold nor with node Accepted.

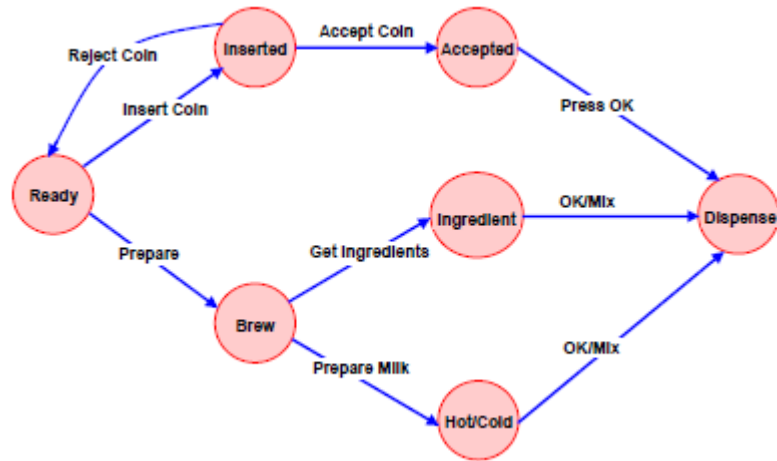


Fig. 3.8: Partial order semantics implied by a beverage selling machine

Merits of formal requirements specification

Formal methods possess several positive features, some of which are discussed below.

- Formal specifications encourage rigour. Often, the very process of construction of a rigorous specification is more important than the formal specification itself. The construction of a rigorous specification clarifies several aspects of system behavior that are not obvious in an informal specification.
- Formal methods usually have a well-founded mathematical basis. Thus, formal specifications are not only more precise, but also mathematically sound and can be used to reason about the properties of a specification and to rigorously prove that an implementation satisfies its specifications.
- Formal methods have well-defined semantics. Therefore, ambiguity in specifications is automatically avoided when one formally specifies a system.
- The mathematical basis of the formal methods facilitates automating the analysis of specifications. For example, a tableau-based technique has been used to automatically check the consistency of specifications. Also, automatic theorem proving techniques can be used to verify that an implementation satisfies its specifications. The possibility of automatic verification is one of the most important advantages of formal methods.

- Formal specifications can be executed to obtain immediate feedback on the features of the specified system. This concept of executable specifications is related to rapid prototyping. Informally, a prototype is a “toy” working model of a system that can provide immediate feedback on the behavior of the specified system, and is especially useful in checking the completeness of specifications.

Limitations of formal requirements specification

It is clear that formal methods provide mathematically sound frameworks within large, complex systems can be specified, developed and verified in a systematic rather than in an ad hoc manner. However, formal methods suffer from several shortcomings, some of which are the following:

- Formal methods are difficult to learn and use.
- The basic incompleteness results of first-order logic suggest that it is impossible to check absolute correctness of systems using theorem proving techniques.
- Formal techniques are not able to handle complex problems. This shortcoming results from the fact that, even moderately complicated problems blow up the complexity of formal specification and their analysis. Also, a large unstructured set of mathematical formulas is difficult to comprehend.

3.2.4 Axiomatic specification

In axiomatic specification of a system, first-order logic is used to write the pre and post-conditions to specify the operations of the system in the form of axioms. The pre-conditions basically capture the conditions that must be satisfied before an operation can successfully be invoked. In essence, the pre-conditions capture the requirements on the input parameters of a function. The post-conditions are the conditions that must be satisfied when a function completes execution for the function to be considered to have executed successfully. Thus, the post-conditions are essentially constraints on the results produced for the function execution to be considered successful.

The following are the sequence of steps that can be followed to systematically develop the axiomatic specifications of a function:

- Establish the range of input values over which the function should behave correctly. Also find out other constraints on the input parameters and write it in the form of a predicate.
- Specify a predicate defining the conditions which must hold on the output of the function if it behaved properly.
- Establish the changes made to the function's input parameters after execution of the function. Pure mathematical functions do not change their input and therefore this type of assertion is not necessary for pure functions.
- Combine all of the above into pre and post conditions of the function.

Example1

Specify the pre- and post-conditions of a function that takes a real number as argument and returns half the input value if the input is less than or equal to 100, or else returns double the value.

$f(x : \text{real}) : \text{real}$

pre : $x \in \mathbb{R}$

post : $\{(x \leq 100) \wedge (f(x) = x/2)\} \vee \{(x > 100) \wedge (f(x) = 2*x)\}$

Example 2

Axiomatically specify a function named search which takes an integer array and an integer key value as its arguments and returns the index in the array where the key value is present.

$\text{search}(X : \text{IntArray}, \text{key} : \text{Integer}) : \text{Integer}$

pre : $\exists i \in [X_{\text{first}} \dots X_{\text{last}}], X[i] = \text{key}$

post : $\{(X'[\text{search}(X, \text{key})] = \text{key}) \wedge (X = X')\}$

Here the convention followed is: If a function changes any of its input parameters and if that parameter is named X, and then it is referred to as X' after the function completes execution much faster.

3.3 ALGEBRAIC SPECIFICATION

In the algebraic specification technique, an object class or type is specified in terms of relationships existing between the operations defined on that type. It was first brought into prominence by Guttag [1980, 1985] in specification of abstract

data types. Various notations of algebraic specifications have evolved, including those based on OBJ and Larch languages.

3.3.1 Representation of algebraic specification

Essentially, algebraic specifications define a system as a heterogeneous algebra. A heterogeneous algebra is a collection of different sets on which several operations are defined. Traditional algebras are homogeneous. A homogeneous algebra consists of a single set and several operations; $\{I, +, -, *, /\}$. In contrast, alphabetic strings together with operations of concatenation and length $\{A, I, \text{con}, \text{len}\}$, is not a homogeneous algebra, since the range of the length operation is the set of integers. Each set of symbols in the algebra, in turn, is called a *sort* of the algebra. To define a heterogeneous algebra, we first need to specify its signature, the involved operations, and their domains and ranges. Using algebraic specification, we define the meaning of a set of interface procedure by using equations. An algebraic specification is usually presented in four sections.

Types section

In this section, the sorts (or the data types) being used is specified.

Exceptions section

This section gives the names of the exceptional conditions that might occur when different operations are carried out. These exception conditions are used in the later sections of an algebraic specification. For example, in a queue, possible exceptions are novalue (empty queue), underflow (removal from an empty queue), etc.

Syntax section

This section defines the signatures of the interface procedures. The collection of sets that form input domain of an operator and the sort where the output is produced are called the signature of the operator. For example, the append operation takes a queue and an element and returns a new queue. This is represented as: *append : queue x element* \rightarrow *queue*

Equations section

This section gives a set of rewrite rules (or equations) defining the meaning of the interface procedures in terms of each other. In general, this section is allowed to

contain conditional expressions. For example, a rewrite rule to identify an empty queue may be written as:

isempty(create()) = true

By convention each equation is implicitly universally quantified over all possible values of the variables. Names not mentioned in the syntax section such as ‘r’ or ‘e’ are variables. The first step in defining an algebraic specification is to identify the set of required operations. After having identified the required operators, it is helpful to classify them as either basic constructor operators, extra constructor operators, basic inspector operators, or extra inspection operators. The definition of these categories of operators is as follows:

1. **Basic construction operators.** These operators are used to create or modify entities of a type. The basic construction operators are essential to generate all possible element of the type being specified. For example, ‘create’ and ‘append’ are basic construction operators for a FIFO *queue*.
2. **Extra construction operators.** These are the construction operators other than the basic construction operators. For example, the operator ‘remove’ is an extra construction operator for a FIFO queue because even without using ‘remove’, it is possible to generate all values of the type being specified.
3. **Basic inspection operators.** These operators evaluate attributes of a type without modifying them, e.g., eval, get, etc. Let S be the set of operators whose range is not the data type being specified. The set of the basic operators S1 is a subset of S, such that each operator from S-S1 can be expressed in terms of the operators from S1. For example, ‘isempty’ is a basic inspection operator because it does not modify the FIFO queue type.
4. **Extra inspection operators.** These are the inspection operators that are not basic inspectors.

A good rule of thumb while writing an algebraic specification is to first establish the constructor (basic and extra) are and inspection operators (basic and extra). Then write down an axiom for composition of each basic construction operator over each basic inspection operator and extra constructor operator. Also, write down an axiom for each of the extra inspector in terms of any of the basic inspectors. Thus, if there are m1 basic constructors, m2 extra constructors, n1 basic inspectors, and n2 extra inspectors, we should have $m1 \times (m2+n1) + n2$

axioms are the minimum required and many more axioms may be needed to make the specification complete. Using a complete set of rewrite rules, it is possible to simplify an arbitrary sequence of operations on the interface procedures.

Develop algebraic specification of simple problems

The first step in defining an algebraic specification is to identify the set of required operations. After having identified the required operators, it is helpful to classify them into different categories.

A simple way to determine whether an operator is a constructor (basic or extra) or an inspector (basic or extra) is to check the syntax expression for the operator. If the type being specified appears on the right hand side of the expression then it is a constructor, otherwise it is an inspection operator. For example, in a FIFO queue, 'create' is a constructor because the data type specified 'queue' appears on the right hand side of the expression. But, 'first' and 'is empty' are inspection operators since they do not modify the *queue* data type.

Example

Let us specify a FIFO queue supporting the operations *create*, *append*, *remove*, *first*, and *isempty* where the operations have their usual meaning.

Types:

defines queue

uses boolean, integer

Exceptions:

underflow, novalue

Syntax:

1. $create : \varphi \rightarrow queue$
2. $append : queue \times element \rightarrow queue$
3. $remove : queue \rightarrow queue + \{underflow\}$
4. $first : queue \rightarrow element + \{novalue\}$
5. $isempty : queue \rightarrow boolean$

Equations:

1. $isempty(create()) = true$
2. $isempty(append(q,e)) = false$
3. $first(create()) = novalue$
4. $first(append(q,e)) = is\ isempty(q)\ then\ e\ else\ first(q)$

5. $remove(create()) = underflow$

6. $remove(append(q,e)) = if\ isempty(q)\ then\ create()\ else\ append(remove(q),e)$

In this example, there are two basic constructors (*create* and *append*), one extra construction operator (*remove*) and two basic inspectors (*first* and *empty*). Therefore, there are $2 \times (1+2) + 0 = 6$ equations.

3.3.2 Properties of algebraic specifications

Three important properties that every algebraic specification should possess are:

- **Completeness:** This property ensures that using the equations, it should be possible to reduce any arbitrary sequence of operations on the interface procedures. There is no simple procedure to ensure that an algebraic specification is complete.
- **Finite termination property:** This property essentially addresses the following question: Do applications of the rewrite rules to arbitrary expressions involving the interface procedures always terminate? For arbitrary algebraic equations, convergence (finite termination) is undecidable. But, if the right hand side of each rewrite rule has fewer terms than the left, then the rewrite process must terminate.
- **Unique termination property:** This property indicates whether application of rewrite rules in different orders always result in the same answer. Essentially, to determine this property, the answer to the following question needs to be checked: Can all possible sequence of choices in application of the rewrite rules to an arbitrary expression involving the interface procedures always give the same number? Checking the unique termination property is a very difficult problem.

3.3.3 Structured specification

Developing algebraic specifications is time consuming. Therefore, efforts have been made to devise ways to ease the task of developing algebraic specifications. The following are some of the techniques that have successfully been used to reduce the effort in writing the specifications.

- **Incremental specification.** The idea behind incremental specification is to first develop the specifications of the simple types and then specify more complex types by using the specifications of the simple types.
- **Specification instantiation.** This involves taking an existing specification which has been developed using a generic parameter and instantiating it with some other sort.

3.3.4 Advantages and disadvantages of algebraic specifications

Algebraic specifications have a strong mathematical basis and can be viewed as heterogeneous algebra. Therefore, they are unambiguous and precise. Using an algebraic specification, the effect of any arbitrary sequence of operations involving the interface procedures can automatically be studied. A major shortcoming of algebraic specifications is that they cannot deal with side effects. Therefore, algebraic specifications are difficult to interchange with typical programming languages. Also, algebraic specifications are hard to understand.

3.3.5 Executable specification language (4GLs)

If the specification of a system is expressed formally or by using a programming language, then it becomes possible to directly execute the specification. However, executable specifications are usually slow and inefficient, 4GLs³ (4th Generation Languages) are examples of executable specification languages. 4GLs are successful because there is a lot of commonality across data processing applications. 4GLs rely on software reuse, where the common abstractions have been identified and parameterized. Careful experiments have shown that rewriting 4GL programs in higher level languages results in up to 50% lower memory usage and also the program execution time can reduce ten folds. Example of a 4GL is Structured Query Language (SQL).

Check Your Progress II

1. Functional requirements address maintainability, portability, and usability issues. State True or False.
2. The edges of decision tree represent corresponding actions to be performed according to conditions. State True or False.

3. The upper rows of the decision table specify the corresponding actions to be taken when an evaluation test is satisfied. State True or False.
4. A column in a decision table is called an attribute. State True or False.
5. Pre – conditions of axiomatic specifications state the requirements on the parameters of the function before the function can start executing. State True or False.
6. Post – conditions of axiomatic specifications state the requirements on the parameters of the function when the function is completed. State True or False.
7. Homogeneous algebra is a collection of different sets on which several operations are defined. State True or False.
8. Applications developed using 4 GLs would normally be more efficient and run faster compared to applications developed using 3 GL. State True or False.

3.4 SUMMARY

1. The system analyst starts requirements gathering and analysis activity by collecting all information from the customer which could be used to develop the requirements of the system.
2. The system analyst then analyzes the collected information to obtain a clear and thorough understanding of the product to be developed, with a view to removing all ambiguities and inconsistencies from the initial customer perception of the problem.
3. After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various requirements problems. The most important requirements problems that the analyst has to identify and eliminate are the problems of anomalies, inconsistencies, and incompleteness.
4. When the analyst detects any inconsistencies, anomalies or incompleteness in the gathered requirements, he resolves them by carrying out further discussions with the end-users and the customers.
5. The functional requirements part discusses the functionalities required from the system.

6. Nonfunctional requirements deal with the characteristics of the system which cannot be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.
7. The goals of implementation part documents some general suggestions regarding development. These suggestions guide trade-off among design goals.
8. The goals of implementation section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future, reusability issues, etc.
9. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.
10. A formal specification technique is a mathematical method to specify a hardware and/or software system, verify whether a specification is realizable, verify that an implementation satisfies its specification, prove properties of a system without necessarily running the system, etc.
11. The mathematical basis of a formal method is provided by the specification language.
12. Formal methods are usually classified into two broad categories: model-oriented and property-oriented approaches. In a model-oriented style, one defines a system's behavior directly by constructing a model of the system in terms of mathematical structures such as tuples, relations, functions, sets, sequences, etc. In the property-oriented style, the system's behavior is defined indirectly by stating its properties, usually in the form of a set of axioms that the system must satisfy.

3.6 MODEL QUESTIONS

1. Identify at least four roles of a system analyst.
2. Identify three important parts of an SRS document.
3. Without developing an SRS document an organization might face severe problems. Identify those problems.
4. Identify the non-functional requirement-issues that are considered for any given problem description?

5. Mention at least five problems that an unstructured specification would create during software development.
6. Identify the necessity of using formal specification technique in the context of requirements specification.
7. Identify at least two disadvantages of formal technique.
8. Identify at least two differences between model-oriented and property-oriented approaches in the context of requirements specification.
9. Explain the use of operational semantic.
10. Identify the requirements of algebraic specifications in order to define a system.
11. Identify the use of algebraic specifications in the context of requirements specification.
12. Identify at least two properties of a structured specification.
13. Identify at least two advantages of algebraic specification.
14. Identify at least two disadvantages of algebraic specification.
15. Write down at least two features of an executable specification language with examples.

Answers to Check Your Progress I

1. a, d and e
2. c
3. b

Answers to Check Your Progress II

1. False
2. False
3. False
4. False
5. True
6. True
7. False
8. False

UNIT IV: SOFTWARE DESIGN ISSUES

4.0 LEARNING OBJECTIVES

At the end of this lesson the student will be able to:

- Identify the software design activities.
- Identify the items to be designed during the preliminary and detailed design activities.
- Identify the primary differences between analysis and design activities
- Identify the important items developed during the software design phase.
- State the important desirable characteristics of a good software design.
- Identify the necessary features of a design document in order to facilitate understandability.
- State what cohesion means.
- Classify the different types of cohesion that a module may possess.
- State what coupling means.
- Classify the different types of coupling between modules.
- State when a module can be called functionally independent of other modules.
- State why functional independence is the key factor for a good software design.
- State the salient features of a function-oriented design approach.
- State the salient features of an object-oriented design approach.
- Differentiate between function-oriented and object-oriented design approach.

4.1 BASIC CONCEPTS IN SOFTWARE DESIGN

4.1.1 Software design and its activities

Software design deals with transforming the customer requirements, as described in the SRS document, into a form (a set of documents) that is suitable for

implementation in a programming language. A good software design is seldom arrived by using a single step procedure but rather through several iterations through a series of steps. Design activities can be broadly classified into two important parts:

- Preliminary (or high-level) design, and
- Detailed design.

Preliminary and detailed design activities

The meaning and scope of two design activities (i.e. high-level and detailed design) tend to vary considerably from one methodology to another. High-level design means identification of different modules and the control relationships among them and the definition of the interfaces among these modules. The outcome of high-level design is called the program structure or software architecture. Many different types of notations have been used to represent a high-level design. A popular way is to use a tree-like diagram called the structure chart to represent the control hierarchy in a high-level design. However, other notations such as Jackson diagram [1975] or Warnier-Orr [1977, 1981] diagram can also be used. During detailed design, the data structure and the algorithms of the different modules are designed. The outcome of the detailed design stage is usually known as the module-specification document.

Difference between analysis and design

The aim of analysis is to understand the problem with a view to eliminate any deficiencies in the requirement specification such as incompleteness, inconsistencies, etc. The model which we are trying to build may be or may not be ready. The aim of design is to produce a model that will provide a seamless transition to the coding phase, i.e. once the requirements are analyzed and found to be satisfactory, a design model is created which can be easily implemented.

Items developed during the software design phase

For a design to be easily implemented in a conventional programming language, the following items must be designed during the design phase.

- Different modules required to implement the design solution.
- Control relationship among the identified modules. The relationship is also known as the call relationship or invocation relationship among modules.

- Interface among different modules. The interface among different modules identifies the exact data items exchanged among the modules.
- Data structures of the individual modules.
- Algorithms required implementing each individual module.

Characteristics of a good software design

The definition of “a good software design” can vary depending on the application being designed. For example, the memory size used by a program may be an important issue to characterize a good solution for embedded software development – since embedded applications are often required to be implemented using memory of limited size due to cost, space, or power consumption considerations. For embedded applications, one may sacrifice design comprehensibility to achieve code compactness. For embedded applications, factors like design comprehensibility may take a back seat while judging the goodness of design. Therefore, the criteria used to judge how good a given design solution is can vary widely depending upon the application. Not only is the goodness of design dependent on the targeted application, but also the notion of goodness of a design itself varies widely across software engineers and academicians. However, most researchers and software engineers agree on a few desirable characteristics that every good software design for general application must possess. The characteristics are listed below:

- **Correctness:** A good design should correctly implement all the functionalities identified in the SRS document.
- **Understandability:** A good design is easily understandable.
- **Efficiency:** It should be efficient.
- **Maintainability:** It should be easily amenable to change.

Possibly the most important goodness criterion is design correctness. A design has to be correct to be acceptable. Given that a design solution is correct, understandability of a design is possibly the most important issue to be considered while judging the goodness of a design. A design that is easy to understand is also easy to develop, maintain and change. Thus, unless a design is easily understandable, it would require tremendous effort to implement and maintain it.

Features of a design document

In order to facilitate understandability, the design should have the following features:

- It should use consistent and meaningful names for various design components.
- The design should be modular. The term modularity means that it should use a cleanly decomposed set of modules.
- It should neatly arrange the modules in a hierarchy, e.g. in a tree-like diagram.

Check Your Progress I

For the following, mark all options which are true.

1. The desirable characteristics that every good software design needs are
 - a. Correctness
 - b. Understandability
 - c. Efficiency
 - d. Maintainability
 - e. All of the above
2. A module is said to have logical cohesion, if
 - a. it performs a set of tasks that relate to each other very loosely.
 - b. all the functions of the module are executed within the same time span.
 - c. all elements of the module perform similar operations, e.g. error handling, data input, data output etc.
 - d. None of the above.

Mark the following as either True or False.

3. Coupling between two modules is nothing but a measure of the degree of dependence between them.
4. The primary characteristic of a good design is low cohesion and high coupling.
5. A module having high cohesion and low coupling is said to be functionally independent of other modules.
6. The degree of coupling between two modules does not depend on their interface complexity.

4.2. AN OVERVIEW OF CURRENT DESIGN APPROACHES

4.2.1 Cohesion

Most researchers and engineers agree that a good software design implies clean decomposition of the problem into modules, and the neat arrangement of these modules in a hierarchy. The primary characteristics of neat module decomposition are high cohesion and low coupling. Cohesion is a measure of functional strength of a module. A module having high cohesion and low coupling is said to be functionally independent of other modules. By the term functional independence, we mean that a cohesive module performs a single task or function. A functionally independent module has minimal interaction with other modules.

4.2.1.1 Classification of cohesion

The different classes of cohesion that a module may possess are depicted in fig. 4.1.

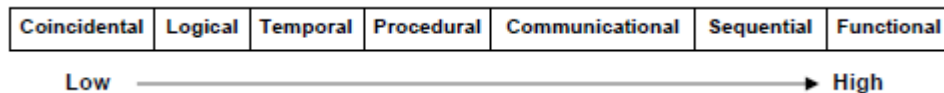


Fig. 4.1: Classification of cohesion

- **Coincidental cohesion:** A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, the module contains a random collection of functions. It is likely that the functions have been put in the module out of pure coincidence without any thought or design. For example, in a transaction processing system (TPS), the get-input, print-error, and summarize-members functions are grouped into one module. The grouping does not have any relevance to the structure of the problem.
- **Logical cohesion:** A module is said to be logically cohesive, if all elements of the module perform similar operations, e.g. error handling, data input, data output, etc. An example of logical cohesion is the case

where a set of print functions generating different output reports are arranged into a single module.

- **Temporal cohesion:** When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion. The set of functions responsible for initialization, start-up, shutdown of some process, etc. exhibit temporal cohesion.
- **Procedural cohesion:** A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective, e.g. the algorithm for decoding a message.
- **Communicational cohesion:** A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure, e.g. the set of functions defined on an array or a stack.
- **Sequential cohesion:** A module is said to possess sequential cohesion, if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next. For example, in a TPS, the get-input, validate-input, sort-input functions are grouped into one module.
- **Functional cohesion:** Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function. For example, a module containing all the functions required to manage employees' payroll exhibits functional cohesion. Suppose a module exhibits functional cohesion and we are asked to describe what the module does, then we would be able to describe it using a single sentence.

4.2.2 Coupling

Coupling between two modules is a measure of the degree of interdependence or interaction between the two modules. A module having high cohesion and low coupling is said to be functionally independent of other modules. If two modules interchange large amounts of data, then they are highly interdependent. The degree of coupling between two modules depends on their interface complexity. The

interface complexity is basically determined by the number of types of parameters that are interchanged while invoking the functions of the module.

4.2.2.1 Classification of Coupling

Even if there are no techniques to precisely and quantitatively estimate the coupling between two modules, classification of the different types of coupling will help to quantitatively estimate the degree of coupling between two modules. Five types of coupling can occur between any two modules. This is shown in fig. 4.2.

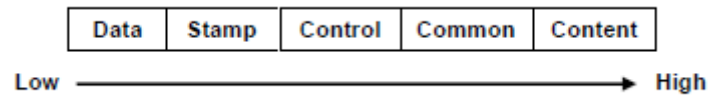


Fig. 4.2: Classification of coupling

Data coupling: Two modules are data coupled, if they communicate through a parameter. An example is an elementary data item passed as a parameter between two modules, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for the control purpose.

Stamp coupling: Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

Control coupling: Control coupling exists between two modules, if data from one module is used to direct the order of instructions execution in another. An example of control coupling is a flag set in one module and tested in another module.

Common coupling: Two modules are common coupled, if they share data through some global data items.

Content coupling: Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module.

4.2.3 Functional independence

A module having high cohesion and low coupling is said to be functionally independent of other modules. By the term functional independence, we mean that a cohesive module performs a single task or function. A functionally independent module has minimal interaction with other modules.

4.2.3.1 Need for functional independence

Functional independence is a key to any good design due to the following reasons:

- **Error isolation:** Functional independence reduces error propagation. The reason behind this is that if a module is functionally independent, its degree of interaction with the other modules is less. Therefore, any error existing in a module would not directly effect the other modules.
- **Scope of reuse:** Reuse of a module becomes possible. Because each module does some well-defined and precise function, and the interaction of the module with the other modules is simple and minimal. Therefore, a cohesive module can be easily taken out and reused in a different program.
- **Understandability:** Complexity of the design is reduced, because different modules can be understood in isolation as modules are more or less independent of each other.

4.2.3.2 Function-oriented design

The following are the salient features of a typical function-oriented design approach:

1. A system is viewed as something that performs a set of functions. Starting at this high-level view of the system, each function is successively refined into more detailed functions. For example, consider a function create-new-library-member which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge. This function may consist of the following sub-functions:

- assign-membership-number
- create-member-record
- print-bill

Each of these sub-functions may be split into more detailed sub functions and so on.

2. The system state is centralized and shared among different functions, e.g. data such as member-records is available for reference and updation to several functions such as:

- create-new-member
- delete-member

- update-member-record

4.2.3.3 Object-oriented design

In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities). The state is decentralized among the objects and each object manages its own state information. For example, in a Library Automation Software, each library member may be a separate object with its own data and functions to operate on these data. In fact, the functions defined for one object cannot refer or change data of other objects. Objects have their own internal data which define their state. Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from super class. Conceptually, objects communicate by message passing.

4.2.3.4 Function-oriented vs. object-oriented design approach

The following are some of the important differences between function-oriented and object-oriented design.

- Unlike function-oriented design methods, in OOD, the basic abstraction are not real-world functions such as sort, display, track, etc, but real-world entities such as employee, picture, machine, radar system, etc. For example in OOD, an employee pay-roll software is not developed by designing functions such as update-employee-record, get-employee-address, etc. but by designing objects such as employees, departments, etc. Grady Booch sums up this difference as “identify verbs if you are after procedural design and nouns if you are after object-oriented design”
- In OOD, state information is not represented in a centralized shared memory but is distributed among the objects of the system. For example, while developing an employee pay-roll system, the employee data such as the names of the employees, their code numbers, basic salaries, etc. are usually implemented as global data in a traditional programming system; whereas in an object-oriented system these data are distributed among different employee objects of the system. Objects communicate by message passing. Therefore, one object may discover the state information of another object by interrogating it. Of course, somewhere or other the

real-world functions must be implemented. In OOD, the functions are usually associated with specific real-world entities (objects); they directly access only part of the system state information.

- Function-oriented techniques such as SA/SD group functions together if, as a group, they constitute a higher-level function. On the other hand, object-oriented techniques group functions together on the basis of the data they operate on.

To illustrate the differences between the object-oriented and the function-oriented design approaches, an example can be considered.

4.2.3.5 Example: Fire-Alarm System

The owner of a large multi-stored building wants to have a computerized fire alarm system for his building. Smoke detectors and fire alarms would be placed in each room of the building. The fire alarm system would monitor the status of these smoke detectors. Whenever a fire condition is reported by any of the smoke detectors, the fire alarm system should determine the location at which the fire condition is reported by any of the smoke detectors, the fire alarm system should determine the location at which the fire condition has occurred and then sound the alarms only in the neighboring locations. The fire alarm system should also flash an alarm message on the computer console. Fire fighting personnel man the console round the clock. After a fire condition has been successfully handled, the fire alarm system should support resetting the alarms by the firefighting personnel.

Function-Oriented Approach

```
/* Global data (system state) accessible by various functions */  
BOOL detector_status[MAX_ROOMS]; int detector_locs[MAX_ROOMS];  
BOOL alarm_status[MAX_ROOMS];  
/* alarm activated when status is set */ int alarm_locs[MAX_ROOMS];  
/* room number where alarm is located */  
int neighbor-alarm[MAX_ROOMS][10];  
/* each detector has at most 10 neighboring locations */  
The functions which operate on the system state are: interrogate_detectors();  
get_detector_location(); determine_neighbor(); ring_alarm(); reset_alarm();  
report_fire_location();
```

Object-Oriented Approach

class detector attributes:

status, location, neighbors

operations:

create, sense_status, get_location, find_neighbors

class alarm attributes:

location, status

operations:

create, ring_alarm, get_location, reset_alarm

In the object-oriented program, an appropriate number of instances of the class detector and alarm should be created. If the function-oriented and the object-oriented programs are examined, it can be seen that in the function-oriented program, the system state is centralized and several functions accessing this central data are defined. In case of the object-oriented program, the state information is distributed among various sensor and alarm objects.

It is not necessary an object-oriented design be implemented by using an object-oriented language only. However, an object-oriented language such as C++ supports the definition of all the basic mechanisms of class, inheritance, objects, methods, etc. and also support all key object-oriented concepts that we have just discussed. Thus, an object-oriented language facilitates the implementation of an OOD. However, an OOD can as well be implemented using a conventional procedural language – though it may require more effort to implement an OOD using a procedural language as compared to the effort required for implementing the same design using an object-oriented language.

Even though object-oriented and function-oriented approaches are remarkably different approaches to software design, yet they do not replace each other but complement each other in some sense. For example, usually one applies the top-down function-oriented techniques to design the internal methods of a class, once the classes are identified. In this case, though outwardly the system appears to

have been developed in an object-oriented fashion, but inside each class there may be a small hierarchy of functions designed in a top-down manner.

Check Your Progress II

1. High coupling among modules makes it
 - a. difficult to understand and maintain the product
 - b. difficult to implement and debug
 - c. expensive to develop the product as the modules having high coupling cannot be developed independently
 - d. all of the above
2. Functional independence results in
 - a. error isolation
 - b. scope of reuse
 - c. understandability
 - d. all of the above

Mark the following as either True or False.

3. In the function-oriented design approach, the system state is decentralized and not shared among different functions.
4. The essence of any good function-oriented design technique is to map the functions performing similar activities into a module.
5. In the object-oriented design, the basic abstraction is real-world functions.
6. An OOD (Object-Oriented Design) can be implemented using object-oriented languages only.

4.3 SUMMARY

1. Coupling between two modules is a measure of the degree of interdependence or interaction between the two modules.
2. Neat module decomposition of a design problem into modules means that the modules in a software design should display high cohesion and low coupling. Conceptually it means that the modules in a design solution are more or less independent of each other.
3. By the term functional independence, it is meant that a cohesive module performs a single task or function.

4. A functionally independent module has minimal interaction with other modules.
5. In a good design, the module should have high cohesion, when similar functions (e.g. print) are put into a module, it displays logical cohesion however functional cohesion is the best cohesion.
6. The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language.
7. A good software design is seldom arrived by using a single step procedure but rather through several iterations through a series of steps.
8. High-level design means identification of different modules and the control relationships among them and the definition of the interfaces among these modules.
9. The outcome of high-level design is called the program structure or software architecture.
10. During detailed design, the data structure and the algorithms of the different modules are designed.
11. The outcome of the detailed design stage is usually known as the module-specification document.
12. Functional independence reduces error propagation. The reason behind this is that if a module is functionally independent, its degree of interaction with the other modules is less. Therefore, any error existing in a module would not directly effect the other modules.
13. Reuse of a module becomes possible. Because each module does some well-defined and precise function and the interaction of the module with the other modules is simple and minimal. Therefore, a cohesive module can be easily taken out and reused in a different program.
14. Complexity of the design is reduced, because different modules can be understood in isolation as modules are more or less independent of each other.
15. In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities).

16. The state is decentralized among the objects and each object manages its own state information. For example, in a Library Automation Software, each library member may be a separate object with its own data and functions to operate on these data. In fact, the functions defined for one object cannot refer or change data of other objects.
17. Objects have their own internal data which define their state.
18. Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from super class. Conceptually, objects communicate by message passing.

4.4 MODEL QUESTIONS

1. Identify at least five important items developed during software design phase.
2. State two major design activities.
3. Identify at least two activities undertaken during high-level design.
4. Identify at least two activities undertaken during detailed design.
5. Identify at least three reasons in favor of why functional independence is the key factor for a good software design.
6. Identify four characteristics of a good software design technique.
7. Identify at least two salient features of a function-oriented design approach.
8. Identify at three least salient features of an object-oriented design approach.
9. Write down at least three differences between function-oriented and object-oriented design approach.

Answers to Check Your Progress I

1. e
2. c
3. d
4. d
5. False
6. False
7. True
8. False

Answers to Check Your Progress II

1. d
2. d
3. False
4. False
5. False
6. False

UNIT V: FUNCTION-ORIENTED SOFTWARE DESIGN

5.0 LEARNING OBJECTIVES

At the end of this lesson the student will be able to:

- Identify the activities carried out during the structured analysis phase.
- Explain what a DFD is.
- Explain why constructing DFDs are important in arriving at a good software design.
- Explain what a data dictionary is.
- Explain the importance of data dictionary.
- Identify whether a DFD is balanced.
- Draw the context diagram of any given problem.
- Draw the DFD model of any given problem.
- Develop the data dictionary for any given problem.
- Identify the common errors that may occur while constructing the DFD model of a system.
- Identify the shortcomings of a DFD model when used as a tool for structured analysis.

5.1 DATA FLOW DIAGRAMS (DFDS)

5.1.1 Structured Analysis

Structured analysis is used to carry out the top-down decomposition of a set of high-level functions depicted in the problem description and to represent them graphically. During structured analysis, functional decomposition of the system is achieved. That is, each function that the system performs is analyzed and hierarchically decomposed into more detailed functions. Structured analysis technique is based on the following essential underlying principles:

- Top-down decomposition approach.

- Divide and conquer principle. Each function is decomposed independently.
- Graphical representation of the analysis results using Data Flow Diagrams (DFDs).

5.1.2 Data Flow Diagram (DFD)

The DFD (also known as a bubble chart) is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among these functions. Each function is considered as a processing station (or process) that consumes some input data and produces some output data. The system is represented in terms of the input data to the system, various processing carried out on these data, and the output data generated by the system. A DFD model uses a very limited number of primitive symbols [as shown in fig. 5.1(a)] to represent the functions performed by a system and the data flow among these functions.

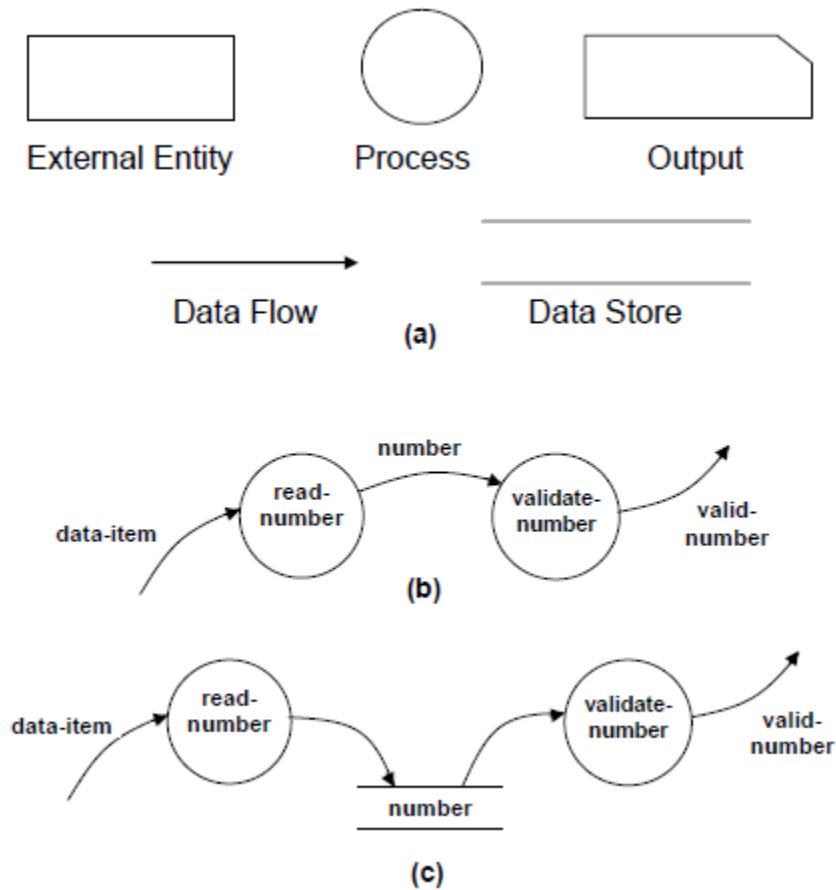


Fig. 5.1 (a) Symbols used for designing DFDs
(b), (c) Synchronous and asynchronous data flow

Here, two examples of data flow that describe input and validation of data are considered. In Fig. 5.1(b), the two processes are directly connected by a data flow. This means that the ‘validate-number’ process can start only after the ‘read-number’ process had supplied data to it. However in Fig 5.1(c), the two processes are connected through a data store. Hence, the operations of the two bubbles are independent. The first one is termed ‘synchronous’ and the second one ‘asynchronous’.

5.1.2.1 Importance of DFDs in a good software design

The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism – it is simple to understand and use. Starting with a set of high-level functions that a system performs, a DFD model hierarchically represents various sub-functions. In fact, any hierarchical model is simple to understand. Human mind is such that it can easily understand any hierarchical model of a system – because in a hierarchical model, starting with a very simple and abstract model of a system, different details of the system are slowly introduced through different hierarchies. The data flow diagramming technique also follows a very simple set of intuitive concepts and rules. DFD is an elegant modeling technique that turns out to be useful not only to represent the results of structured analysis of a software problem, but also for several other applications such as showing the flow of documents or items in an organization.

5.1.3 Data dictionary

A data dictionary lists all data items appearing in the DFD model of a system. The data items listed include all data flows and the contents of all data stores appearing on the DFDs in the DFD model of a system. A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items. For example, a data dictionary entry may represent that the data **grossPay** consists of the components **regularPay** and **overtimePay**.

$$\mathbf{grossPay = regularPay + overtimePay}$$

For the smallest units of data items, the data dictionary lists their name and their type. Composite data items can be defined in terms of primitive data items using the following data definition operators:

+: denotes composition of two data items, e.g. **a+b** represents data **a** and **b**.

[,,]: represents selection, i.e. any one of the data items listed in the brackets can occur. For example, **[a,b]** represents either **a** occurs or **b** occurs.

(): the contents inside the bracket represent optional data which may or may not appear. e.g. **a+(b)** represents either **a** occurs or **a+b** occurs.

{}: represents iterative data definition, e.g. **{name}5** represents five **name** data.
{name}* represents zero or more instances of **name** data.

=: represents equivalence, e.g. **a=b+c** means that **a** represents **b** and **c**.

/* */: Anything appearing within **/*** and ***/** is considered as a comment.

Example 1: Tic-Tac-Toe Computer Game

Tic-tac-toe is a computer game in which a human player and the computer make alternative moves on a 3×3 square. A move consists of marking previously unmarked square. The player who first places three consecutive marks along a straight line on the square (i.e. along a row, column, or diagonal) wins the game. As soon as either the human player or the computer wins, a message congratulating the winner should be displayed. If neither player manages to get three consecutive marks along a straight line, but all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.

It may be recalled that the DFD model of a system typically consists of several DFDs: level 0, level 1, etc. However, a single data dictionary should capture all the data appearing in all the DFDs constituting the model. Figure 5.2 represents the level 0 and level 1 DFDs for the tic-tac-toe game.

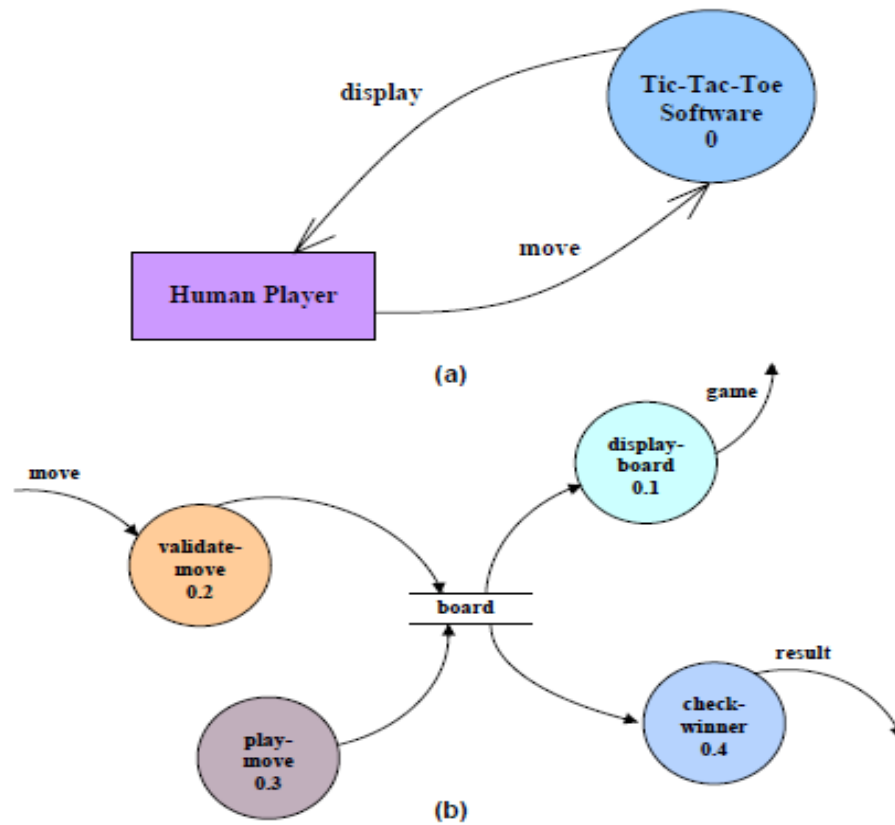


Fig 5.2 (a), (b) Level 0 and Level 1 DFD for Tic-Tac-Toe game described in Example 1

The data dictionary for the model is given below.

Data dictionary for the DFD model in Example 1

- move:** integer /*number between 1 and 9 */
- display:** game+result
- game:** board
- board:** {integer}9
- result:** ["computer won", "human won" "draw"]

5.1.3.1 Importance of data dictionary

A data dictionary plays a very important role in any software development process because of the following reasons:

- A data dictionary provides a standard terminology for all relevant data for use by the engineers working in a project. A consistent vocabulary for data items is very important, since in large projects different engineers of the project have

a tendency to use different terms to refer to the same data, which unnecessary causes confusion.

- The data dictionary provides the analyst with a means to determine the definition of different data structures in terms of their component elements.

5.1.4 Balancing a DFD

The data that flow into or out of a bubble must match the data flow at the next level of DFD. This is known as balancing a DFD. The concept of balancing a DFD has been illustrated in fig. 5.3. In the level 1 of the DFD, data items d1 and d3 flow out of the bubble 0.1 and the data item d2 flows into the bubble 0.1. In the next level, bubble 0.1 is decomposed. The decomposition is balanced, as d1 and d3 flow out of the level 2 diagram and d2 flows in.

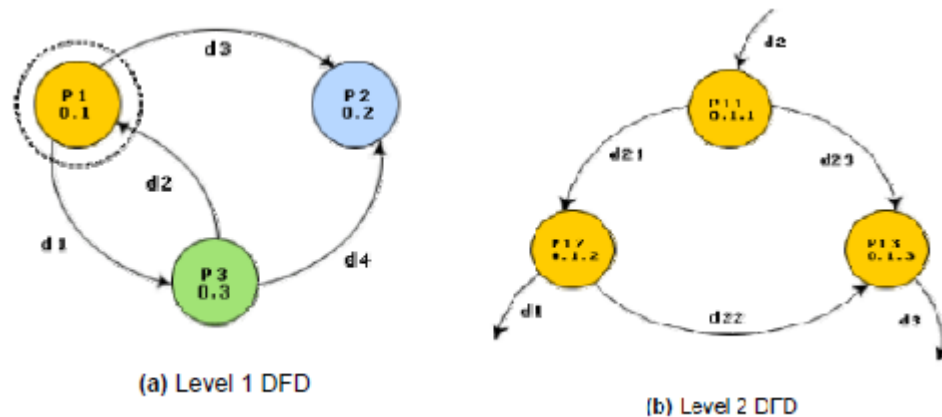


Fig. 5.3: An example showing balanced decomposition

Check Your Progress I

For the following, mark all options which are true.

1. The purpose of structured analysis is
 - a. to capture the detailed structure of the system as perceived by the user
 - b. to define the structure of the solution that is suitable for implementation in some programming language
 - c. all of the above
2. Structured analysis technique is based on
 - a. top-down decomposition approach
 - b. bottom-up approach

- c. divide and conquer principle
 - d. none of the above
3. Data Flow Diagram (DFD) is also known as a:
- a. structure chart
 - b. bubble chart
 - c. Gantt chart
 - d. PERT chart
4. The context diagram of a DFD is also known as
- a. level 0 DFD
 - b. level 1 DFD
 - c. level 2 DFD
 - d. none of the above
5. A DFD model of a system represents the functions performed by the system and the data flow taking place among these functions. (State True of False)
6. A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items. (State True of False)

5.2 DFD MODEL OF A SYSTEM

5.2.1 Context diagram

The context diagram is the most abstract data flow representation of a system. It represents the entire system as a single bubble. This bubble is labeled according to the main function of the system. The various external entities with which the system interacts and the data flow occurring between the system and the external entities are also represented. The data input to the system and the data output from the system are represented as incoming and outgoing arrows. These data flow arrows should be annotated with the corresponding data names. The name 'context diagram' is well justified because it represents the context in which the system is to exist, i.e. the external entities who would interact with the system and the specific data items they would be supplying the system and the data items they

would be receiving from the system. The context diagram is also called as the level 0 DFD.

To develop the context diagram of the system, it is required to analyze the SRS document to identify the different types of users who would be using the system and the kinds of data they would be inputting to the system and the data they would be receiving the system. Here, the term “users of the system” also includes the external systems which supply data to or receive data from the system.

The bubble in the context diagram is annotated with the name of the software system being developed (usually a noun). This is in contrast with the bubbles in all other levels which are annotated with verbs. This is expected since the purpose of the context diagram is to capture the context of the system rather than its functionality.

Example#1: RMS Calculating Software.

A software system called RMS calculating software would read three integral numbers from the user in the range of -1000 and +1000 and then determine the root mean square (rms) of the three input numbers and display it. In this example, the context diagram (fig. 5.4) is simple to draw. The system accepts three integers from the user and returns the result to him.

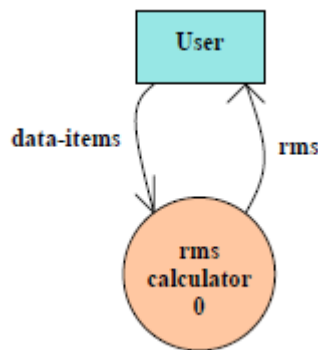


Fig. 5.4: Context Diagram

Example#2: Tic-Tac-Toe Computer Game

The problem is described in Lesson 5.1(Example 1). The level 0 DFD shown in Figure 5.2(a) is the context diagram for this problem.

5.2.2 DFD model of a system

A DFD model of a system graphically depicts the transformation of the data input to the system to the final result through a hierarchy of levels. A DFD starts with the most abstract definition of the system (lowest level) and at each higher level DFD, more details are successively introduced. To develop a higher-level DFD model, processes are decomposed into their sub-processes and the data flow among these sub-processes is identified.

To develop the data flow model of a system, first the most abstract representation of the problem is to be worked out. The most abstract representation of the problem is also called the context diagram. After, developing the context diagram, the higher-level DFDs have to be developed.

Context Diagram

This has been described earlier.

Level 1 DFD

To develop the level 1 DFD, examine the high-level functional requirements. If there are between 3 to 7 high-level functional requirements, then these can be directly represented as bubbles in the level 1 DFD. We can then examine the input data to these functions and the data output by these functions and represent them appropriately in the diagram.

If a system has more than 7 high-level functional requirements, then some of the related requirements have to be combined and represented in the form of a bubble in the level 1 DFD. Such a bubble can be split in the lower DFD levels. If a system has less than three high-level functional requirements, then some of them need to be split into their sub-functions so that we have roughly about 5 to 7 bubbles on the diagram.

Decomposition

Each bubble in the DFD represents a function performed by the system. The bubbles are decomposed into sub-functions at the successive levels of the DFD. Decomposition of a bubble is also known as factoring or exploding a bubble. Each bubble at any level of DFD is usually decomposed to anything between 3 to 7 bubbles. Too few bubbles at any level make that level superfluous. For example, if a bubble is decomposed to just one bubble or two bubbles, then this decomposition becomes redundant. Also, too many bubbles, i.e. more than 7

bubbles at any level of a DFD makes the DFD model hard to understand. Decomposition of a bubble should be carried on until a level is reached at which the function of the bubble can be described using a simple algorithm.

Numbering of Bubbles

It is necessary to number the different bubbles occurring in the DFD. These numbers help in uniquely identifying any bubble in the DFD by its bubble number. The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD. Bubbles at level 1 are numbered, 0.1, 0.2, 0.3, etc, etc. When a bubble numbered x is decomposed, its children bubble are numbered x.1, x.2, x.3, etc. In this numbering scheme, by looking at the number of a bubble we can unambiguously determine its level, its ancestors, and its successors.

Example

A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his/her residence address, telephone number, and the driving license number. Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer. A customer can present his CN to the check out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year. Also, it intends to award a 22 caret gold coin to every customer whose purchase exceeded Rs.10,000. The entries against the CN are the reset on the day of every year after the prize winners' lists are generated. The context diagram for this problem is shown in fig. 5.5, the level 1 DFD in fig. 5.6, and the level 2 DFD in fig. 5.7.

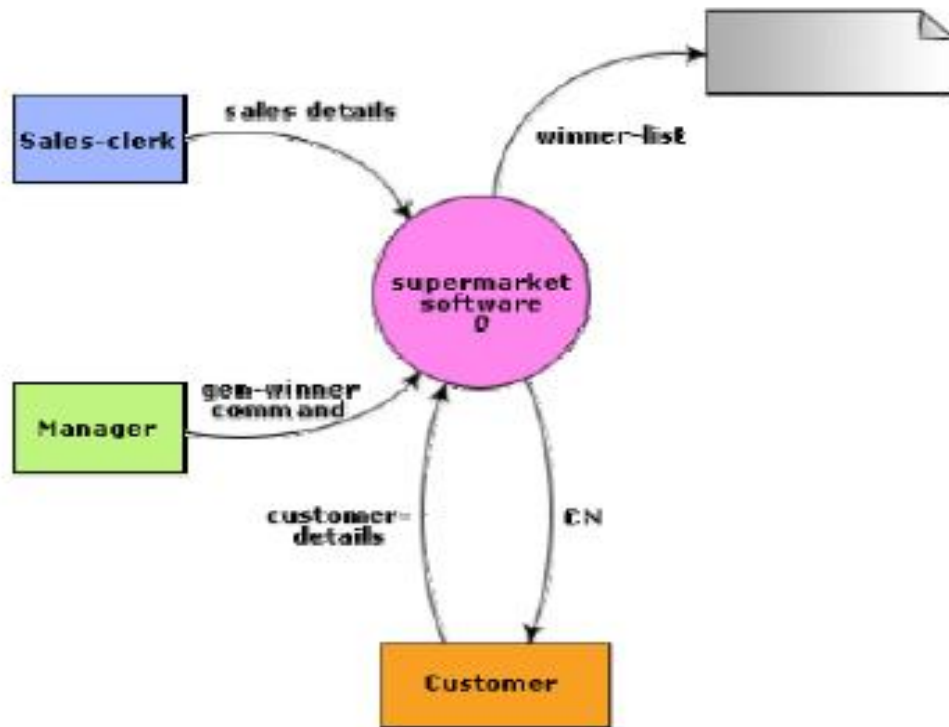


Fig. 5.5: Context diagram for supermarket problem

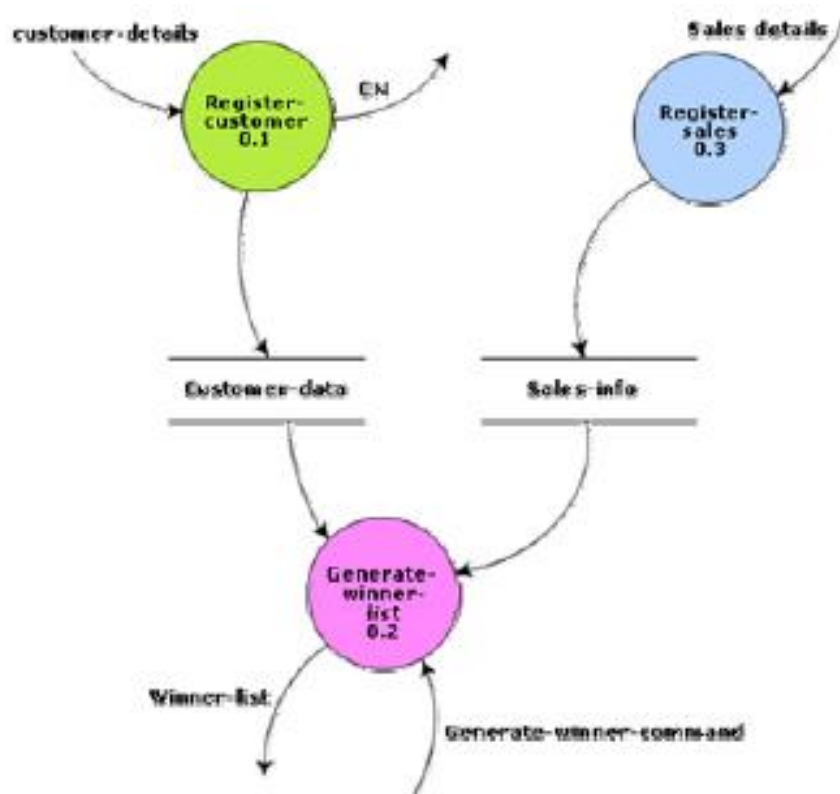


Fig. 5.6: Level 1 diagram for supermarket problem

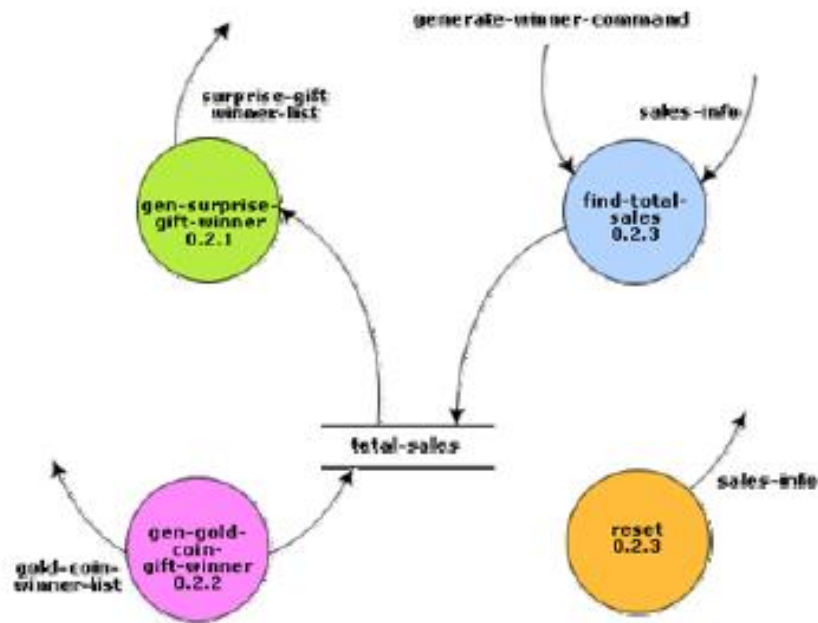


Fig. 5.7: Level 2 diagram for supermarket problem

5.2.3 Data dictionary for a DFD model

Every DFD model of a system must be accompanied by a data dictionary. A data dictionary lists all data items appearing in the DFD model of a system. The data items listed include all data flows and the contents of all data stores appearing on the DFDs in the DFD model of a system. We can understand the creation of a data dictionary better by considering an example.

Example: Trading-House Automation System (TAS).

The trading house wants us to develop a computerized system that would automate various book-keeping activities associated with its business. The following are the salient features of the system to be developed:

- The trading house has a set of regular customers. The customers place orders with it for various kinds of commodities. The trading house maintains the names and addresses of its regular customers. Each of these regular customers should be assigned a unique customer identification number (CIN) by the computer. The customers quote their CIN on every order they place.
- Once order is placed, as per current practice, the accounts department of the trading house first checks the credit-worthiness of the customer. The

credit-worthiness of the customer is determined by analyzing the history of his payments to different bills sent to him in the past. After automation, this task has to be done by the computer.

- If the customer is not credit-worthy, his orders are not processed any further and an appropriate order rejection message is generated for the customer.
- If a customer is credit-worthy, the items that have been ordered are checked against a list of items that the trading house deals with. The items in the order which the trading house does not deal with, are not processed any further and an appropriate apology message for the customer for these items is generated.
- The items in the customer's order that the trading house deals with are checked for availability in the inventory. If the items are available in the inventory in the desired quantity, then
 - A bill with the forwarding address of the customer is printed.
 - A material issue slip is printed. The customer can produce this material issue slip at the store house and take delivery of the items.
 - Inventory data is adjusted to reflect the sale to the customer.
- If any of the ordered items are not available in the inventory in sufficient quantity to satisfy the order, then these out-of-stock items along with the quantity ordered by the customer and the CIN are stored in a "pending-order" file for the further processing to be carried out when the purchase department issues the "generate indent" command.
- The purchase department should be allowed to periodically issue commands to generate indents. When a command to generate indents is issued, the system should examine the "pending-order" file to determine the orders that are pending and determine the total quantity required for each of the items. It should find out the addresses of the vendors who supply these items by examining a file containing vendor details and then should print out indents to these vendors.

- The system should also answer managerial queries regarding the statistics of different items sold over any given period of time and the corresponding quantity sold and the price realized.

The context diagram for the trading house automation problem is shown in fig. 5.8, and the level 1 DFD in fig. 5.9.



Fig. 5.8: Context diagram for TAS

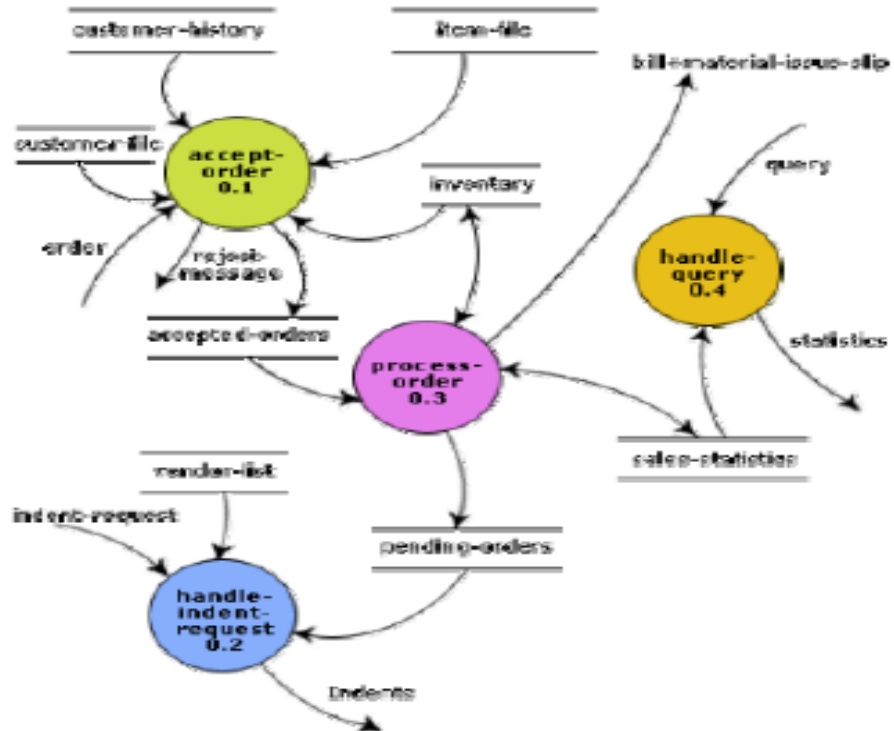


Fig. 5.9: Level 1 DFD for TAS

Data Dictionary for the DFD Model of TAS:

response: [bill + material-issue-slip, reject-message]

query: period /*query from manager regarding sales statistics */

period: [date + date, month, year, day]

date: year + month + day

year: integer

month: integer

day: integer

order: customer-id + {items + quantity}* + order#

accepted-order: order /* ordered items available in inventory */

reject-message: order + message /*rejection message*/

pending-orders: customer-id + {items + quantity}*

customer-address: name + house# + street# + city + pin

name: string

house#: string

street#: string

city: string

pin: integer

customer-id: integer

customer-file: {customer-address}*

bill: {item + quantity + price}* + total-amount + customer-address + order#

material-issue-slip: message + item + quantity + customer-address

message: string

statistics: {item + quantity + price}*

sales-statistics: {statistics}* + date

quantity: integer

order#: integer /* unique order number generated by the program */

price: integer

total-amount: integer

generate-indent: command

indent: {indent + quantity}* + vendor-address

indents: {indent}*

vendor-address: customer-address

vendor-list: {vendor-address}*

item-file: {item}*

item: string

indent-request: command

5.2.4 Commonly made errors while constructing a DFD model

Although DFDs are simple to understand and draw, students and practitioners alike encounter similar types of problems while modelling software problems using DFDs. While learning from experience is a powerful thing, it is an expensive pedagogical technique in the business world. It is therefore helpful to understand the different types of mistakes that users usually make while constructing the DFD model of systems.

- Many beginners commit the mistake of drawing more than one bubble in the context diagram. A context diagram should depict the system as a single bubble.
- Many beginners have external entities appearing at all levels of DFDs. All external entities interacting with the system should be represented only in the context diagram. The external entities should not appear at other levels of the DFD.
- It is a common oversight to have either too less or too many bubbles in a DFD. Only 3 to 7 bubbles per diagram should be allowed, i.e. each bubble should be decomposed to between 3 and 7 bubbles.
- Many beginners leave different levels of DFD unbalanced.
- A common mistake committed by many beginners while developing a DFD model is attempting to represent control information in a DFD. It is important to realize that a DFD is the data flow representation of a system, and it does not represent control information. For an example mistake of this kind:
 - Consider the following example. A book can be searched in the library catalog by inputting its name. If the book is available in the library, then the details of the book are displayed. If the book is not listed in the catalog, then an error message is generated. While generating the DFD model for this simple problem, many beginners

commit the mistake of drawing an arrow (as shown in fig. 5.10) to indicate the error function is invoked after the search book. But, this is a control information and should not be shown on the DFD.

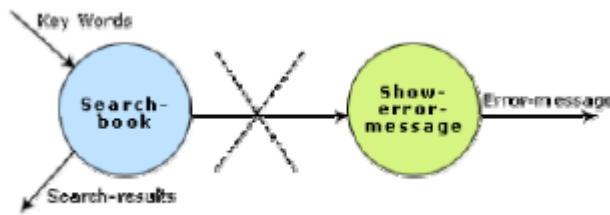


Fig. 5.10: Showing control information on a DFD - incorrect

- o Another error is trying to represent when or in what order different functions (processes) are invoked and not representing the conditions under which different functions are invoked.
 - o If a bubble A invokes either the bubble B or the bubble C depending upon some conditions, we need only to represent the data that flows between bubbles A and B or bubbles A and C and not the conditions depending on which the two modules are invoked.
- A data store should be connected only to bubbles through data arrows. A data store cannot be connected to another data store or to an external entity.
 - All the functionalities of the system must be captured by the DFD model. No function of the system specified in its SRS document should be overlooked.
 - Only those functions of the system specified in the SRS document should be represented, i.e. the designer should not assume functionality of the system not specified by the SRS document and then try to represent them in the DFD.
 - Improper or unsatisfactory data dictionary.
 - The data and function names must be intuitive. Some students and even practicing engineers use symbolic data names such a, b, c, etc. Such names hinder understanding the DFD model.

5.2.5 Shortcomings of a DFD model

DFD models suffer from several shortcomings. The important shortcomings of the DFD models are the following:

- DFDs leave ample scope to be imprecise. In the DFD model, the function performed by a bubble is judged from its label. However, a short label may not capture the entire functionality of a bubble. For example, a bubble named find-book-position has only intuitive meaning and does not specify several things, e.g. what happens when some input information are missing or are incorrect. Further, the find-book-position bubble may not convey anything regarding what happens when the required book is missing.
- Control aspects are not defined by a DFD. For instance, the order in which inputs are consumed and outputs are produced by a bubble is not specified. A DFD model does not specify the order in which the different bubbles are executed. Representation of such aspects is very important for modeling real-time systems.
- The method of carrying out decomposition to arrive at the successive levels and the ultimate level to which decomposition is carried out are highly subjective and depend on the choice and judgment of the analyst. Due to this reason, even for the same problem, several alternative DFD representations are possible. Further, many times it is not possible to say which DFD representation is superior or preferable to another one.
- The data flow diagramming technique does not provide any specific guidance as to how exactly to decompose a given function into its sub-functions and we have to use subjective judgment to carry out decomposition.

Check Your Progress II

1. Decomposition of a bubble is also known as
 - a. classification
 - b. factoring
 - c. exploding
 - d. aggregation
2. Decomposition of a bubble should be carried on

- a. till the atomic program instructions are reached
 - b. upto two levels
 - c. until a level is reached at which the function of the bubble can be described using a simple algorithm
 - d. none of the above
3. The bubbles in a level 1 DFD represent
- a. exactly one high-level functional requirement described in SRS document
 - b. more than one high-level functional requirement
 - c. part of a high-level functional requirement
 - d. any of the above depending on the problem

Mark the following as either True or False. Justify your answer.

4. The context diagram of a system represents it using more than one bubble.
5. External entities may appear at all levels of DFDs.
6. A DFD captures the order in which the processes (bubbles) operate.
7. DFDs enable a software engineer to develop the data domain and functional domain decomposition of the system at the same time.

5.3 SUMMARY

1. The aim of the structured analysis activity is to transform a textual problem description into a graphic model.
2. Structured analysis is used to carry out the top-down decomposition of the set of high-level functions depicted in the problem description and to represent them graphically.
3. During structured analysis, functional decomposition of the system is achieved. That is, each function that the system performs is analyzed and hierarchically decomposed into more detailed functions.
4. During structured analysis, the major processing tasks (functions) of the system are analyzed, and the data flow among those processing tasks is represented graphically.
5. Data Flow Diagram (DFD) is a very simple formalism. It is simple to understand and use.

6. Starting with a set of high-level functions that a system performs, a DFD model hierarchically represents various sub-functions.
7. The data flow diagramming technique follows a very simple set of intuitive concepts and rules.
8. DFD is an elegant modeling technique that turns out to be useful not only to represent the results of structured analysis of a software problem but also useful for several other applications such as showing the flow of documents or items in an organization.
9. A data dictionary plays a very important role in any software development process because it provides a standard terminology for all relevant data for use by the engineers working in a project.
10. A consistent vocabulary for data items is very important, since in large projects, different engineers of the project have a tendency to use different terms to refer to the same data, which unnecessary causes confusion.
11. The data dictionary provides the analyst with a means to determine the definition of different data structures in terms of their component elements.
12. The data that flow into or out of a bubble must match the data flow at the next level of DFD. This is known as balancing a DFD.

5.4 MODEL QUESTIONS

1. Identify the aim of the structured analysis activity. Which documents are produced at the end of structured analysis activity?
2. Identify the necessity of constructing DFDs in the context of a good software design.
3. Write down the importance of data dictionary in the context of good software design.
4. What does the term “balancing a DFD” mean? Give an example to explain your answer.
5. Write down some essential activities required to develop the DFD of a system more systematically.

Answers to Check Your Progress I

1. a
2. a, c
3. b
4. a
5. True
6. True

Answers to Check Your Progress II

1. b, c
2. c
3. d
4. False
5. False
6. False
7. True

UNIT VI: BASIC CONCEPTS IN OBJECT ORIENTATION

6.0 LEARNING OBJECTIVES

At the end of this lesson the student will be able to:

- Identify the aim of structured design.
- Explain what a structure chart is.
- Differentiate between a structure chart and a flow chart.
- Identify the activities carried out during transform analysis with examples.
- Explain what is meant by transaction analysis.

6.1. STRUCTURED DESIGN

6.1.1 Structured Design

The aim of structured design is to transform the results of the structured analysis (i.e. a DFD representation) into a structure chart. Structured design provides two strategies to guide transformation of a DFD into a structure chart.

- Transform analysis
- Transaction analysis

Normally, one starts with the level 1 DFD, transforms it into module representation using either the transform or the transaction analysis and then proceeds towards the lower-level DFDs. At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD. These are discussed in the subsequent subsections.

6.1.1.1 Structure Chart

A structure chart represents the software architecture, i.e. the various modules making up the system, the dependency (which module calls which other modules), and the parameters that are passed among the different modules. Hence, the structure chart representation can be easily implemented using some programming

language. Since the main focus in a structure chart representation is on the module structure of the software and the interactions among different modules, the procedural aspects (e.g. how a particular functionality is achieved) are not represented. The basic building blocks which are used to design structure charts are the following:

- **Rectangular boxes:** Represents a module.
- **Module invocation arrows:** Control is passed from one module to another module in the direction of the connecting arrow.
- **Data flow arrows:** Arrows are annotated with data name; named data passes from one module to another module in the direction of the arrow.
- **Library modules:** Represented by a rectangle with double edges.
- **Selection:** Represented by a diamond symbol.
- **Repetition:** Represented by a loop around the control flow arrow.

6.1.1.2 Structure Chart vs. Flow Chart

We are all familiar with the flow chart representation of a program. Flow chart is a convenient technique to represent the flow of control in a program. A structure chart differs from a flow chart in three principal ways:

- It is usually difficult to identify the different modules of the software from its flow chart representation.
- Data interchange among different modules is not represented in a flow chart.
- Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.

6.2 Transform Analysis

Transform analysis identifies the primary functional components (modules) and the high-level inputs and outputs for these components. The first step in transform analysis is to divide the DFD into 3 types of parts:

- Input
- Logical processing
- Output

The input portion of the DFD includes processes that transform input data from physical (e.g. character from terminal) to logical forms (e.g. internal tables, lists, etc.). Each input portion is called an afferent branch.

The output portion of a DFD transforms output data from logical to physical form. Each output portion is called an efferent branch. The remaining portion of a DFD is called the central transform.

In the next step of transform analysis, the structure chart is derived by drawing one functional component for the central transform, and the afferent and efferent branches. These are drawn below a root module, which would invoke these modules.

Identifying the highest-level input and output transforms requires experience and skill. One possible approach is to trace the inputs until a bubble is found whose output cannot be deduced from its inputs alone. Processes which validate input or add information to them are not central transforms. Processes which sort input or filter data from it are. The first level structure chart is produced by representing each input and output unit as boxes and each central transform as a single box.

In the third step of transform analysis, the structure chart is refined by adding sub-functions required by each of the high-level functional components. Many levels of functional components may be added. This process of breaking functional components into subcomponents is called factoring. Factoring includes adding read and write modules, error-handling modules, initialization and termination process, identifying customer modules, etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

Example: Structure chart for the RMS software

For this example, the context diagram was drawn earlier.

To draw the level 1 DFD (fig. 6.1), from a cursory analysis of the problem description, we can see that there are four basic functions that the system needs to perform – accept the input numbers from the user, validate the numbers, calculate the root mean square of the input numbers and, then display the result.

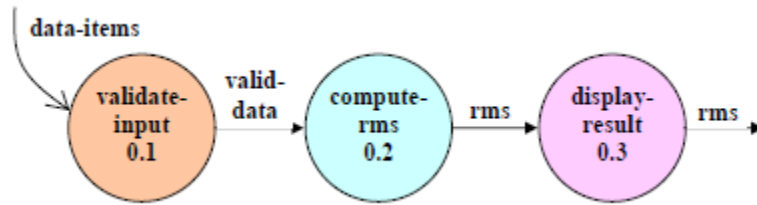


Fig. 6.1: Level 1 DFD

By observing the level 1 DFD, we identify the validate-input as the afferent branch and write-output as the efferent branch. The remaining portion (i.e. compute-rms) forms the central transform. By applying the step 2 and step 3 of transform analysis, we get the structure chart shown in [fig. 6.2](#)

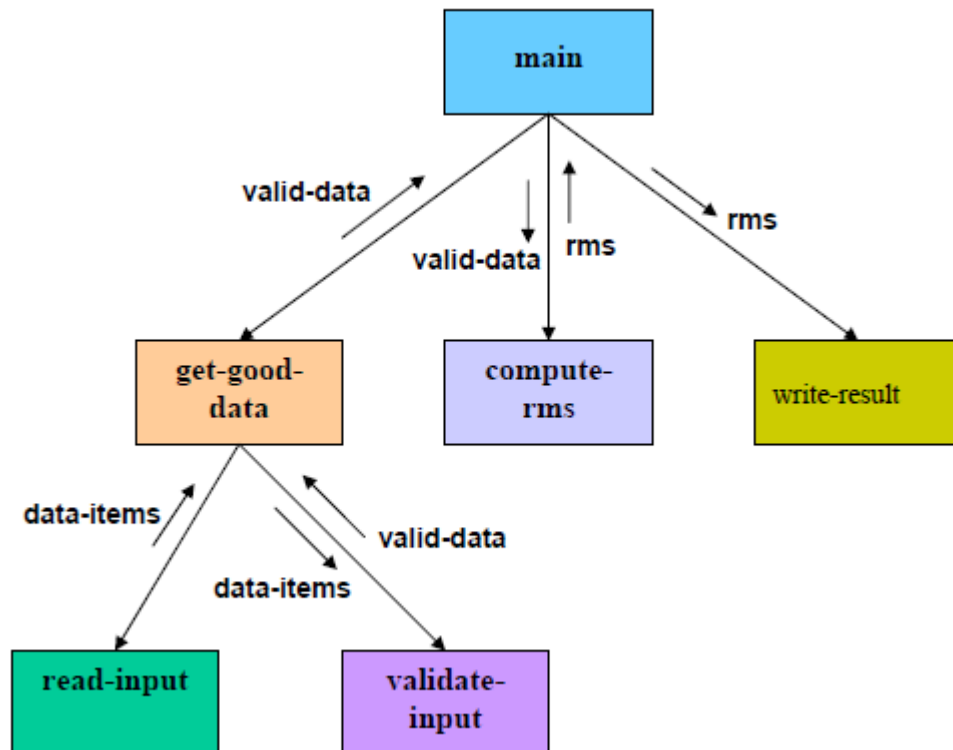


Fig. 6.2: Structure chart

Check Your Progress I

1. By looking at the structure chart, we can
 - a. say whether a module calls another module just once or many times
 - b. not say whether a module calls another module just once or many times
 - c. tell the order in which the different modules are invoked
 - d. not tell the order in which the different modules are invoked

2. In a structure chart, a module represented by a rectangle with double edges is called
 - a. root module
 - b. library module
 - c. primary module
 - d. none of the above
3. A structure chart differs from a flow chart in which of the following ways
 - a. it is always difficult to identify the different modules of the software from its flow chart representation
 - b. data interchange among different modules is not presented in a flow chart
 - c. sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart
 - d. none of the above
4. The input portion in the DFD that transform input data from physical to logical form is called
 - a. central transform
 - b. efferent branch
 - c. afferent branch
 - d. none of the above
5. If during structured design you observe that the data entering a DFD are incident on different bubbles, then you would use:
 - a. transform analysis
 - b. transaction analysis
 - c. combination of transform and transaction analysis
 - d. neither transform nor transaction analysis

6.2.1 Transaction Analysis

A transaction allows the user to perform some meaningful piece of work. Transaction analysis is useful while designing transaction processing programs. In a transaction-driven system, one of several possible paths through the DFD is traversed depending upon the input data item. This is in contrast to a transform centered system which is characterized by similar processing steps for each data

item. Each different way in which input data is handled is a transaction. A simple way to identify a transaction is to check the input data. The number of bubbles on which the input data to the DFD are incident defines the number of transactions. However, some transaction may not require any input data. These transactions can be identified from the experience of solving a large number of examples.

For each identified transaction, trace the input data to the output. All the traversed bubbles belong to the transaction. These bubbles should be mapped to the same module on the structure chart. In the structure chart, draw a root module and below this module draw each identified transaction a module. Every transaction carries a tag, which identifies its type. Transaction analysis uses this tag to divide the system into transaction modules and a transaction-center module.

The structure chart for the supermarket prize scheme software is shown in fig. 6.3.

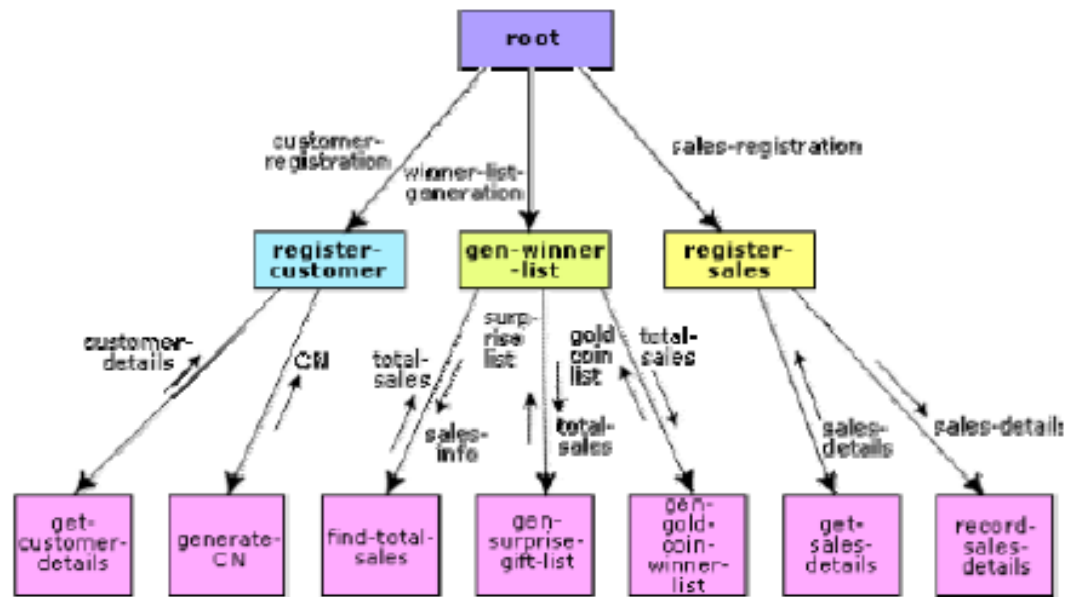


Fig. 6.3: Structure chart for the supermarket prize scheme

Check Your Progress II

1. During structured design, if all the data flow into the diagram are processed in similar ways i.e. if all the input data are incident on the same bubble in the DFD, the one have to use:
 - a. transform analysis
 - b. transaction analysis
 - c. combination of transform and transaction analysis

- d. neither transform nor transaction analysis
2. Which of the following types of bubbles may belong to the central transform?
 - a. input validation
 - b. adding information to the input
 - c. sorting input ✓
 - d. filtering data
 3. During detailed design which of the following activities take place?
 - a. the pseudo code for the different modules of the structure chart are developed in the form of MSPECs
 - b. data structures are designed for the different modules of the structure chart
 - c. module structure is designed
 - d. none of the above
 4. There should be at most one control relationship between any two modules in a properly designed structure chart.(State True or False)

6.3 SUMMARY

1. In the context of function-oriented design, top-down decomposition starts with the high-level functional requirements.
2. A structure chart differs from a flow chart in three principal ways:
 - a. It is usually difficult to identify the different modules of the software from its flow chart representation.
 - b. Data interchange among different modules is not represented in a flow chart.
 - c. Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.

6.4 Model Question

1. What do you understand by top-down decomposition in the context of structured analysis? Explain your answer using a suitable example.
2. Identify some commonly made errors while constructing of a DFD model.
3. Identify some important shortcomings of the DFD model.

4. Differentiate between a structure chart and a flow chart.

Answers to Check Your Progress I

1. b,d
2. b
3. a,b,c
4. c
5. b

Answers to Check Your Progress II

1. a
2. d
3. a,b
4. True

UNIT VII: OBJECT MODELING USING UML

7.0 LEARNING OBJECTIVES

At the end of this lesson the student will be able to:

- Explain what a model is.
- Explain how models are useful.
- Explain what UML means.
- Explain the origin and the acceptance of UML in industry.
- Identify different types of views captured by UML diagrams.
- Identify different use cases of a system.
- Identify the purpose of use cases.
- Represent use cases for a particular system.
- Explain the utility of the use case diagram.
- Factorize use cases into different component use cases.
- Explain the organization of use cases.
- Explain the features represented by a class diagram.
- Explain the relationships among different types of classes by means of association.
- Explain the relationships among different types of classes by means of aggregation.
- Explain the relationships among different types of classes by means of composition.
- Draw interaction diagrams for any given problem.
- Explain the tradeoff between inheritance and aggregation/ composition
- Bring out a comparison of the three relationships: association, aggregation and composition
- Draw activity diagrams for any given problem.
- Differentiate between the activity diagrams and procedural flow charts.
- Develop the state chart diagram for any given class.
- Compare activity diagrams with state chart diagrams.

7.1. BASIC IDEAS ON UML

7.1.1 Model

A model captures aspects important for some application while omitting (or abstracting) the rest. A model in the context of software development can be graphical, textual, mathematical, or program code-based. Models are very useful in documenting the design and analysis results. Models also facilitate the analysis and design procedures themselves. Graphical models are very popular because they are easy to understand and construct. UML is primarily a graphical modeling tool. However, it often requires text explanations to accompany the graphical models.

7.1.1.1 Need for a model

An important reason behind constructing a model is that it helps manage complexity. Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the following:

- Analysis
- Specification
- Code generation
- Design
- Visualize and understand the problem and the working of a system
- Testing, etc.

In all these applications, the UML models can not only be used to document the results but also to arrive at the results themselves. Since a model can be used for a variety of purposes, it is reasonable to expect that the model would vary depending on the purpose for which it is being constructed. For example, a model developed for initial analysis and specification should be very different from the one used for design. A model that is being used for analysis and specification would not show any of the design decisions that would be made later on during the design stage. On the other hand, a model used for design purposes should capture all the design decisions. Therefore, it is a good idea to explicitly mention the purpose for which a model has been developed, along with the model.

7.1.2 Unified Modeling Language (UML)

UML, as the name implies, is a modeling language. It may be used to visualize, specify, construct, and document the artifacts of a software system. It provides a set of notations (e.g. rectangles, lines, ellipses, etc.) to create a visual model of the system. Like any other language, UML has its own syntax (symbols and sentence formation rules) and semantics (meanings of symbols and sentences). Also, we should clearly understand that UML is not a system design or development methodology, but can be used to document object-oriented and analysis results obtained using some methodology.

7.1.2.1 Origin of UML

In the late 1980s and early 1990s, there was a proliferation of object-oriented design techniques and notations. Different software development houses were using different notations to document their object-oriented designs. These diverse notations used to give rise to a lot of confusion.

UML was developed to standardize the large number of object-oriented modeling notations that existed and were used extensively in the early 1990s. The principles ones in use were:

- Object Management Technology [Rumbaugh 1991]
- Booch's methodology [Booch 1991]
- Object-Oriented Software Engineering [Jacobson 1992]
- Odell's methodology [Odell 1992]
- Shaler and Mellor methodology [Shaler 1992]

It is needless to say that UML has borrowed many concepts from these modeling techniques. Especially, concepts from the first three methodologies have been heavily drawn upon. UML was adopted by Object Management Group (OMG) as a *de facto* standard in 1997. OMG is an association of industries which tries to facilitate early formation of standards.

We shall see that UML contains an extensive set of notations and suggests construction of many types of diagrams. It has successfully been used to model both large and small problems. The elegance of UML, its adoption by OMG, and a strong industry backing have helped UML find widespread acceptance. UML is now being used in a large number of software development projects worldwide.

7.1.2.2 UML diagrams

UML can be used to construct nine different types of diagrams to capture five different views of a system. Just as a building can be modeled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.; the different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system.

The UML diagrams can capture the following five views of a system:

- User's view
- Structural view
- Behavioral view
- Implementation view
- Environmental view

Fig. 7.1 shows the UML diagrams responsible for providing the different views.

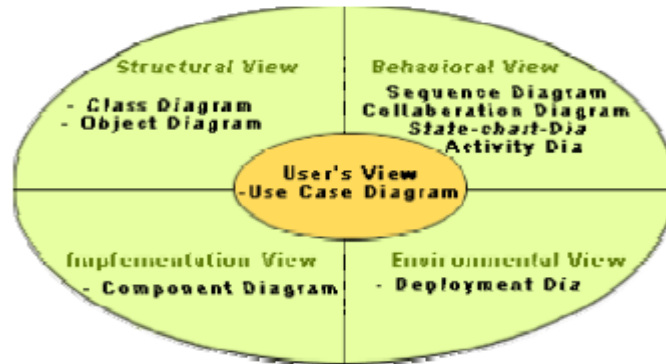


Fig. 7.1: Different types of diagrams and views supported in UML

User's view: This view defines the functionalities (facilities) made available by the system to its users. The users' view captures the external users' view of the system in terms of the functionalities offered by the system. The users' view is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible. The users' view is very different from all other views in the sense that it is a functional model compared to the object model of all other views. The users' view can be

considered as the central view and all other views are expected to conform to this view. This thinking is in fact the crux of any user centric development style.

Structural view: The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects). The structural model is also called the static model, since the structure of a system does not change with time.

Behavioral view: The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.

Implementation view: This view captures the important components of the system and their dependencies.

Environmental view: This view models how the different components are implemented on different pieces of hardware.

7.2 USE CASE MODEL

The use case model for any system consists of a set of “use cases”. Intuitively, use cases represent the different ways in which a system can be used by the users. A simple way to find all the use cases of a system is to ask the question: “What the users can do using the system?”. Thus, for the Library Information System (LIS), the use cases could be:

- issue-book
- query-book
- return-book
- create-member
- add-book, etc

Use cases correspond to the high-level functional requirements. The use cases partition the system behavior into transactions, such that each transaction performs some useful action from the user’s point of view. To complete each transaction may involve either a single message or multiple message exchanges between the user and the system to complete.

7.2.1 Purpose of use cases

The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the system. The use cases do not mention any specific algorithm to be used or the internal data representation, internal structure of the software, etc. A use case typically represents a sequence of interactions between the user and the system. These interactions consist of one mainline sequence. The mainline sequence represents the normal interaction between a user and the system. The mainline sequence is the most occurring sequence of interaction. For example, the mainline sequence of the withdraw cash use case supported by a bank ATM drawn, complete the transaction, and get the amount. Several variations to the main line sequence may also exist. Typically, a variation from the mainline sequence occurs when some specific conditions hold. For the bank ATM example, variations or alternate scenarios may occur, if the password is invalid or the amount to be withdrawn exceeds the amount balance. The variations are also called alternative paths. A use case can be viewed as a set of related scenarios tied together by a common goal. The mainline sequence and each of the variations are called scenarios or instances of the use case. Each scenario is a single path of user events and system activity through the use case.

7.2.2 Representation of use cases

Use cases can be represented by drawing a use case diagram and writing an accompanying text elaborating the drawing. In the use case diagram, each use case is represented by an ellipse with the name of the use case written inside the ellipse. All the ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary. The name of the system being modeled (such as Library Information System) appears inside the rectangle.

The different users of the system are represented by using the stick person icon. Each stick person icon is normally referred to as an actor. An actor is a role played by a user with respect to the system use. It is possible that the same user may play the role of multiple actors. Each actor can participate in one or more use cases. The line connecting the actor and the use case is called the communication relationship. It indicates that the actor makes use of the functionality provided by the use case. Both the human users and the external systems can be represented by

stick person icons. When a stick person icon represents an external system, it is annotated by the stereotype <<external system>>.

Example 1:

The use case model for the Tic-tac-toe problem is shown in fig. 7.2. This software has only one use case “play move”. Note that the use case “get-user-move” is not used here. The name “get-user-move” would be inappropriate because the use cases should be named from the users’ perspective.

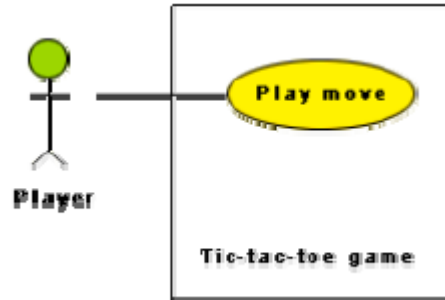


Fig. 7.2: Use case model for tic-tac-toe game

Text Description

Each ellipse on the use case diagram should be accompanied by a text description. The text description should define the details of the interaction between the user and the computer and other aspects of the use case. It should include all the behavior associated with the use case in terms of the mainline sequence, different variations to the normal behavior, the system responses associated with the use case, the exceptional conditions that may occur in the behavior, etc. The behavior description is often written in a conversational style describing the interactions between the actor and the system. The text description may be informal, but some structuring is recommended. The following are some of the information which may be included in a use case text description in addition to the mainline sequence, and the alternative scenarios.

Contact persons: This section lists the personnel of the client organization with whom the use case was discussed, date and time of the meeting, etc.

Actors: In addition to identifying the actors, some information about actors using this use case which may help the implementation of the use case may be recorded.

Pre-condition: The preconditions would describe the state of the system before the use case execution starts.

Post-condition: This captures the state of the system after the use case has successfully completed.

Non-functional requirements: This could contain the important constraints for the design and implementation, such as platform and environment conditions, qualitative statements, response time requirements, etc.

Exceptions, error situations: This contains only the domain-related errors such as lack of user’s access rights, invalid entry in the input fields, etc. Obviously, errors that are not domain related, such as software errors, need not be discussed here.

Sample dialogs: These serve as examples illustrating the use case.

Specific user interface requirements: These contain specific requirements for the user interface of the use case. For example, it may contain forms to be used, screen shots, interaction style, etc.

Document references: This part contains references to specific domain-related documents which may be useful to understand the system operation.

Example 2:

The use case model for the Supermarket Prize Scheme described in Lesson 5.2 is shown in fig. 7.3. As discussed earlier, the use cases correspond to the high-level functional requirements. From the problem description and the context diagram in fig. 5.5, we can identify three use cases: “register-customer”, “register-sales”, and “select-winners”. As a sample, the text description for the use case “register-customer” is shown.

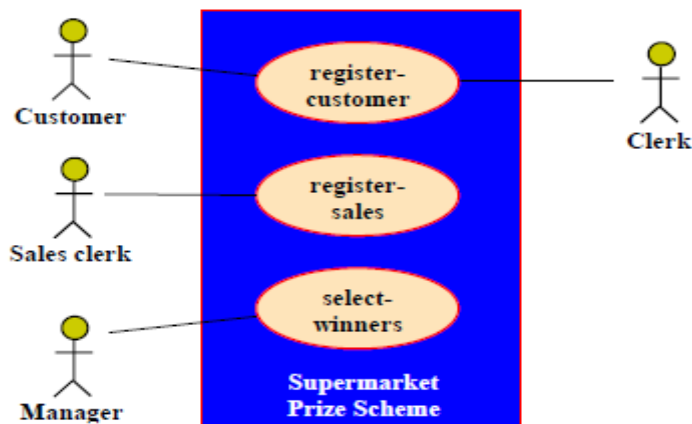


Fig. 7.3 Use case model for Supermarket Prize Scheme

Text description

U1: register-customer: Using this use case, the customer can register himself by providing the necessary details.

Scenario 1: Mainline sequence

- 1. Customer:** select register customer option.
- 2. System:** display prompt to enter name, address, and telephone number.
- 3. Customer:** enter the necessary values.
- 4. System:** display the generated id and the message that the customer has been successfully registered.

Scenario 2: at step 4 of mainline sequence

- 1. System:** displays the message that the customer has already registered.

Scenario 2: at step 4 of mainline sequence

- 1. System:** displays the message that some input information has not been entered. The system displays a prompt to enter the missing value.

The description for other use cases is written in a similar fashion.

7.2.3 Utility of use case diagrams

From use case diagram, it is obvious that the utility of the use cases are represented by ellipses. They along with the accompanying text description serve as a type of requirements specification of the system and form the core model to which all other models must conform. But, what about the actors (stick person icons)? One possible use of identifying the different types of users (actors) is in identifying and implementing a security mechanism through a login system, so that each actor can involve only those functionalities to which he is entitled to. Another possible use is in preparing the documentation (e.g. users' manual) targeted at each category of user. Further, actors help in identifying the use cases and understanding the exact functioning of the system.

7.2.4 Factoring of use cases

It is often desirable to factor use cases into component use cases. Actually, factoring of use cases are required under two situations. First, complex use cases need to be factored into simpler use cases. This would not only make the behavior associated with the use case much more comprehensible, but also make the

corresponding interaction diagrams more tractable. Without decomposition, the interaction diagrams for complex use cases may become too large to be accommodated on a single sized (A4) paper. Secondly, use cases need to be factored whenever there is common behavior across different use cases. Factoring would make it possible to define such behavior only once and reuse it whenever required. It is desirable to factor out common usage such as error handling from a set of use cases. This makes analysis of the class design much simpler and elegant. However, a word of caution here. Factoring of use cases should not be done except for achieving the above two objectives. From the design point of view, it is not advantageous to break up a use case into many smaller parts just for the shake of it.

UML offers three mechanisms for factoring of use cases as follows:

Generalization

Use case generalization can be used when one use case that is similar to another, but does something slightly differently or something more. Generalization works the same way with use cases as it does with classes. The child use case inherits the behavior and meaning of the parent use case. The notation is the same too (as shown in fig. 7.4). It is important to remember that the base and the derived use cases are separate use cases and should have separate text descriptions.

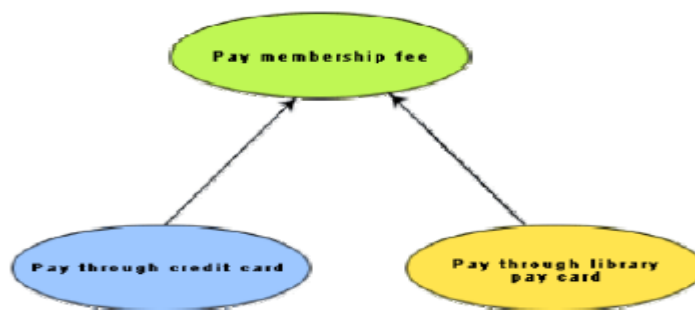


Fig. 7.4: Representation of use case generalization

Includes

The includes relationship in the older versions of UML (prior to UML 1.1) was known as the uses relationship. The includes relationship involves one use case including the behavior of another use case in its sequence of events and actions. The includes relationship occurs when a chunk of behavior that is similar across a

number of use cases. The factoring of such behavior will help in not repeating the specification and implementation across different use cases. Thus, the includes relationship explores the issue of reuse by factoring out the commonality across use cases. It can also be gainfully employed to decompose a large and complex use cases into more manageable parts. As shown in fig. 7.5, the includes relationship is represented using a predefined stereotype <<include>>. In the includes relationship, a base use case compulsorily and automatically includes the behavior of the common use cases. As shown in example fig. 7.6, issue-book and renew-book both include check-reservation use case. The base use case may include several use cases. In such cases, it may interleave their associated common use cases together. The common use case becomes a separate use case and the independent text description should be provided for it.



Fig. 7.5: Representation of use case inclusion

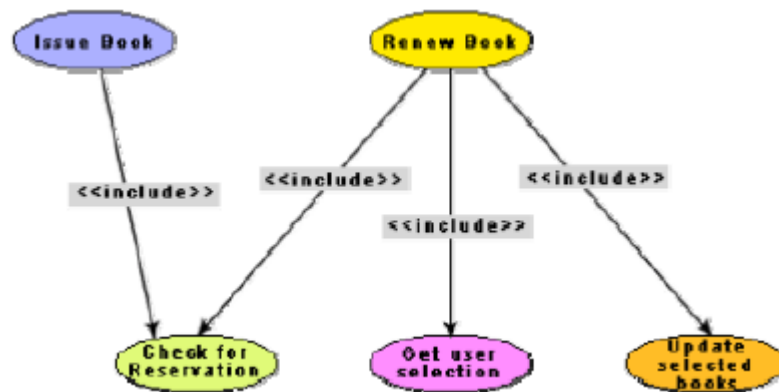


Fig. 7.6: Example use case inclusion

Extends

The main idea behind the extends relationship among the use cases is that it allows you to show optional system behavior. An optional system behavior is extended only under certain conditions. This relationship among use cases is also predefined as a stereotype as shown in fig. 7.7. The extends relationship is similar to

generalization. But unlike generalization, the extending use case can add additional behavior only at an extension point only when certain conditions are satisfied. The extension points are points within the use case where variation to the mainline (normal) action sequence may occur. The extends relationship is normally used to capture alternate paths or scenarios.



Fig. 7.7: Example use case extension

7.2.5 Organization of use cases

When the use cases are factored, they are organized hierarchically. The high-level use cases are refined into a set of smaller and more refined use cases as shown in fig. 7.8. Top-level use cases are super-ordinate to the refined use cases. The refined use cases are sub-ordinate to the top-level use cases. Note that only the complex use cases should be decomposed and organized in a hierarchy. It is not necessary to decompose simple use cases. The functionality of the super-ordinate use cases is traceable to their sub-ordinate use cases. Thus, the functionality provided by the super-ordinate use cases is composite of the functionality of the sub-ordinate use cases. In the highest level of the use case model, only the fundamental use cases are shown. The focus is on the application context. Therefore, this level is also referred to as the context diagram. In the context diagram, the system limits are emphasized. In the top-level diagram, only those use cases with which external users of the system. The subsystem-level use cases specify the services offered by the subsystems. Any number of levels involving the subsystems may be utilized. In the lowest level of the use case hierarchy, the class-level use cases specify the functional fragments or operations offered by the classes.

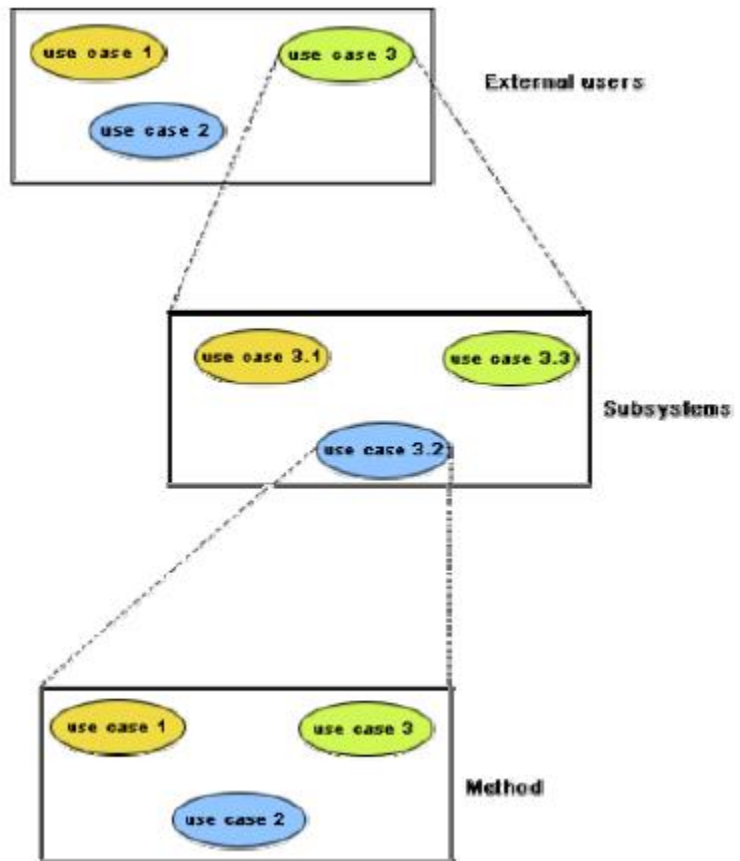


Fig. 7.8: Hierarchical organization of use cases

Check Your Progress I

Mark all options which are true.

1. UML is a
 - a. a language to model syntax
 - b. an object-oriented development methodology
 - c. an automatic code generation tool
 - d. none of the above

2. Which of the following view captured by UML diagrams can be considered as black box model of a system?
 - a. structural view
 - b. behavioral view
 - c. user's view
 - d. environmental view
 - e. implementation view

Mark the following as either True or False. Justify your answer.

3. For any given problem, one should construct all the views using all the diagrams provided by UML.
4. Use cases are explicitly dependent among themselves.
5. Each actor can participate in one and only one use-case.
6. Class diagrams developed using UML can serve as the functional specification of a system.
7. The terms method and operation are equivalent concepts and can be used interchangeably.
8. A class can have an association relationship with itself.
9. The Aggregation relationship can be recursively defined, i.e. an object can contain instances of itself.

7.3 CLASS AND INTERACTION DIAGRAMS

7.3.1 Class diagrams

A class diagram describes the static structure of a system. It shows how a system is structured rather than how it behaves. The static structure of a system comprises of a number of class diagrams and their dependencies. The main constituents of a class diagram are classes and their relationships: generalization, aggregation, association, and various kinds of dependencies.

Classes

The classes represent entities with common features, i.e. attributes and operations. Classes are represented as solid outline rectangles with compartments. Classes have a mandatory name compartment where the name is written centered in boldface. The class name is usually written using mixed case convention and begins with an uppercase. The class names are usually chosen to be singular nouns. An example of a class is shown in fig. 6.2 (Lesson 6.1).

Classes have optional attributes and operations compartments. A class may appear on several diagrams. Its attributes and operations are suppressed on all but one diagram.

Attributes

An attribute is a named property of a class. It represents the kind of data that an object might contain. Attributes are listed with their names, and may optionally contain specification of their type, an initial value, and constraints. The type of the attribute is written by appending a colon and the type name after the attribute name. Typically, the first letter of a class name is a small letter. An example for an attribute is given.

bookName : String

Operation

Operation is the implementation of a service that can be requested from any object of the class to affect behavior. An object's data or state can be changed by invoking an operation of the object. A class may have any number of operations or no operation at all. Typically, the first letter of an operation name is a small letter. Abstract operations are written in italics. The parameters of an operation (if any), may have a kind specified, which may be 'in', 'out' or 'inout'. An operation may have a return type consisting of a single return type expression. An example for an operation is given.

issueBook(in bookName):Boolean

Association

Associations are needed to enable objects to communicate with each other. An association describes a connection between classes. The association relation between two objects is called object connection or link. Links are instances of associations. A link is a physical or conceptual connection between object instances. For example, suppose Amit has borrowed the book Graph Theory. Here, borrowed is the connection between the objects Amit and Graph Theory book. Mathematically, a link can be considered to be a tuple, i.e. an ordered list of object instances. An association describes a group of links with a common structure and common semantics. For example, consider the statement that Library Member borrows Books. Here, borrows is the association between the class Library Member and the class Book. Usually, an association is a binary relation (between two classes). However, three or more different classes can be involved in an association. A class can have an association relationship with itself (called

recursive association). In this case, it is usually assumed that two different objects of the class are linked by the association relationship.

Association between two classes is represented by drawing a straight line between the concerned classes. Fig. 7.9 illustrates the graphical representation of the association relation. The name of the association is written along side the association line. An arrowhead may be placed on the association line to indicate the reading direction of the association. The arrowhead should not be misunderstood to be indicating the direction of a pointer implementing an association. On each side of the association relation, the multiplicity is noted as an individual number or as a value range. The multiplicity indicates how many instances of one class are associated with each other. Value ranges of multiplicity are noted by specifying the minimum and maximum value, separated by two dots, e.g. 1..5. An asterisk is a wild card and means many (zero or more). The association of fig. 7.9 should be read as “Many books may be borrowed by a Library Member”. Observe that associations (and links) appear as verbs in the problem statement.



Fig. 7.9: Association between two classes

Associations are usually realized by assigning appropriate reference attributes to the classes involved. Thus, associations can be implemented using pointers from one object class to another. Links and associations can also be implemented by using a separate class that stores which objects of a class are linked to which objects of another class. Some CASE tools use the role names of the association relation for the corresponding automatically generated attribute.

Aggregation

Aggregation is a special type of association where the involved classes represent a whole-part relationship. The aggregate takes the responsibility of forwarding messages to the appropriate parts. Thus, the aggregate takes the responsibility of delegation and leadership. When an instance of one object contains instances of some other objects, then aggregation (or composition) relationship exists between

the composite object and the component object. Aggregation is represented by the diamond symbol at the composite end of a relationship. The number of instances of the component class aggregated can also be shown as in fig. 7.10.



Fig. 7.10: Representation of aggregation

Aggregation relationship cannot be reflexive (i.e. recursive). That is, an object cannot contain objects of the same class as itself. Also, the aggregation relation is not symmetric. That is, two classes A and B cannot contain instances of each other. However, the aggregation relationship can be transitive. In this case, aggregation may consist of an arbitrary number of levels.

Composition

Composition is a stricter form of aggregation, in which the parts are existence-dependent on the whole. This means that the life of the parts closely ties to the life of the whole. When the whole is created, the parts are created and when the whole is destroyed, the parts are destroyed. A typical example of composition is an invoice object with invoice items. As soon as the invoice object is created, all the invoice items in it are created and as soon as the invoice object is destroyed, all invoice items in it are also destroyed. The composition relationship is represented as a filled diamond drawn at the composite-end. An example of the composition relationship is shown in fig. 7.11.



Fig. 7.11: Representation of composition

7.3.2 Association vs. Aggregation vs. Composition

- Association is the most general (m:n) relationship. Aggregation is a stronger relationship where one is a part of the other. Composition is even

stronger than aggregation, ties the lifecycle of the part and the whole together.

- Association relationship can be reflexive (objects can have relation to itself), but aggregation cannot be reflexive. Moreover, aggregation is anti-symmetric (If B is a part of A, A can not be a part of B).
- Composition has the property of exclusive aggregation i.e. an object can be a part of only one composite at a time. For example, a **Frame** belongs to exactly one **Window** whereas in simple aggregation, a part may be shared by several objects. For example, a **Wall** may be a part of one or more **Room** objects.
- In addition, in composition, the whole has the responsibility for the disposition of all its parts, i.e. for their creation and destruction.
 - in general, the lifetime of parts and composite coincides
 - parts with non-fixed multiplicity may be created after composite itself
 - parts might be explicitly removed before the death of the composite

For example, when a **Frame** is created, it has to be attached to an enclosing **Window**. Similarly, when the **Window** is destroyed, it must in turn destroy its **Frame** parts.

7.3.3 Inheritance vs. Aggregation/Composition

- Inheritance describes *'is a' / 'is a kind of'* relationship between classes (base class - derived class) whereas aggregation describes *'has a'* relationship between classes. Inheritance means that the object of the derived class inherits the properties of the base class; aggregation means that the object of the whole has objects of the part. For example, the relation “cash payment *is a kind of* payment” is modeled using inheritance; “purchase order has a few items” is modeled using aggregation.
- Inheritance is used to model a “generic-specific” relationship between classes whereas aggregation/composition is used to model a “whole-part” relationship between classes.
- Inheritance means that the objects of the subclass can be used anywhere the super class may appear, but not the reverse; i.e. wherever we could use

instances of 'payment' in the system, we could substitute it with instances of 'cash payment', but the reverse can not be done.

- Inheritance is defined statically. It can not be changed at run-time. Aggregation is defined dynamically and can be changed at run-time. Aggregation is used when the type of the object can change over time.
- For example, consider this situation in a business system. A **BusinessPartner** might be a **Customer** or a **Supplier** or both. Initially we might be tempted to model it as in Fig 7.12(a). But in fact, during its lifetime, a business partner might become a customer as well as a supplier, or it might change from one to the other. In such cases, we prefer aggregation instead (see Fig 7.12(b). Here, a business partner is a **Customer** if it has an aggregated **Customer** object, a **Supplier** if it has an aggregated **Supplier** object and a "**Customer_Supplier**" if it has both. Here, we have only two types. Hence, we are able to model it as inheritance. But what if there were several different types and combinations there of? The inheritance tree would be absolutely incomprehensible. Also, the aggregation model allows the possibility for a business partner to be neither - i.e. has neither a customer nor a supplier object aggregated with it.
- The advantage of aggregation is the integrity of encapsulation. The operations of an object are the interfaces of other objects which imply low implementation dependencies. The significant disadvantage of aggregation is the increase in the number of objects and their relationships. On the other hand, inheritance allows for an easy way to modify implementation for reusability. But the significant disadvantage is that it breaks encapsulation, which implies implementation dependence.

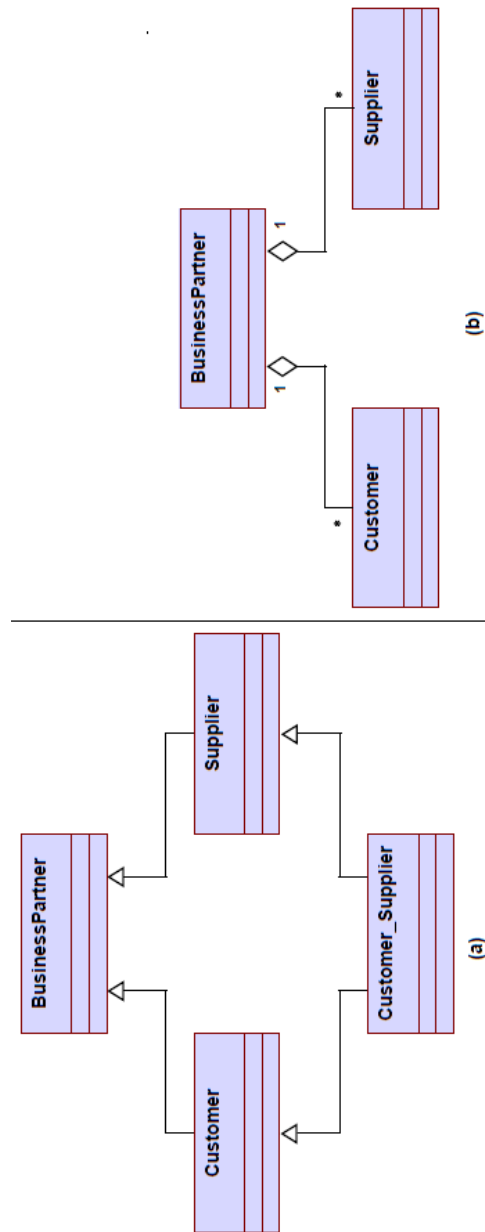


Fig. 7.12 Representation of **BusinessPartner**, **Customer**, **Supplier** relationship
 (a) using inheritance (b) using aggregation

7.3.4 Interaction Diagrams

Interaction diagrams are models that describe how group of objects collaborate to realize some behavior. Typically, each interaction diagram realizes the behavior of a single use case. An interaction diagram shows a number of example objects and the messages that are passed between the objects within the use case.

There are two kinds of interaction diagrams: sequence diagrams and collaboration diagrams. These two diagrams are equivalent in the sense that any one diagram

can be derived automatically from the other. However, they are both useful. These two actually portray different perspectives of behavior of the system and different types of inferences can be drawn from them. The interaction diagrams can be considered as a major tool in the design methodology.

7.3.4.1 Sequence Diagram

A sequence diagram shows interaction among objects as a two-dimensional chart. The chart is read from top to bottom. The objects participating in the interaction are shown at the top of the chart as boxes attached to a vertical dashed line. Inside the box the name of the object is written with a colon separating it from the name of the class and both the name of the object and the class is underlined. The objects appearing at the top signify that the object already existed when the use case execution was initiated. However, if some object is created during the execution of the use case and participates in the interaction (e.g. a method call), then the object should be shown at the appropriate place on the diagram where it is created. The vertical dashed line is called the object's lifeline. The lifeline indicates the existence of the object at any particular point of time. The rectangle drawn on the lifetime is called the activation symbol and indicates that the object is active as long as the rectangle exists. Each message is indicated as an arrow between the lifeline of two objects. The messages are shown in chronological order from the top to the bottom. That is, reading the diagram from the top to the bottom would show the sequence in which the messages occur. Each message is labeled with the message name. Some control information can also be included. Two types of control information are particularly valuable.

- A condition (e.g. [invalid]) indicates that a message is sent, only if the condition is true.
- An iteration marker shows the message is sent many times to multiple receiver objects as would happen when a collection or the elements of an array are being iterated. The basis of the iteration can also be indicated e.g. [for every book object].

The sequence diagram for the book renewal use case for the Library Automation Software is shown in fig. 7.13. The development of the sequence diagram in the

development methodology would help us in determining the responsibilities of the different classes; i.e. what methods should be supported by each class.

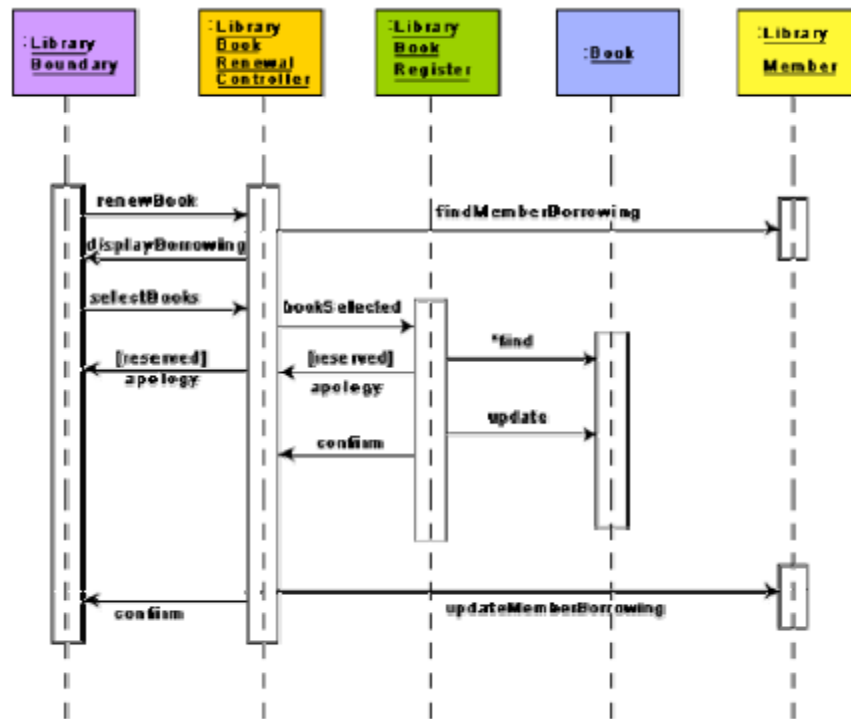


Fig. 7.13: Sequence diagram for the renew book use case

7.3.4.2 Collaboration Diagram

A collaboration diagram shows both structural and behavioral aspects explicitly. This is unlike a sequence diagram which shows only the behavioral aspects. The structural aspect of a collaboration diagram consists of objects and the links existing between them. In this diagram, an object is also called a collaborator. The behavioral aspect is described by the set of messages exchanged among the different collaborators. The link between objects is shown as a solid line and can be used to send messages between two objects. The message is shown as a labeled arrow placed near the link. Messages are prefixed with sequence numbers because they are only way to describe the relative sequencing of the messages in this diagram. The collaboration diagram for the example of fig. 7.13 is shown in fig. 7.14. The use of the collaboration diagrams in our development process would be to help us to determine which classes are associated with which other classes.

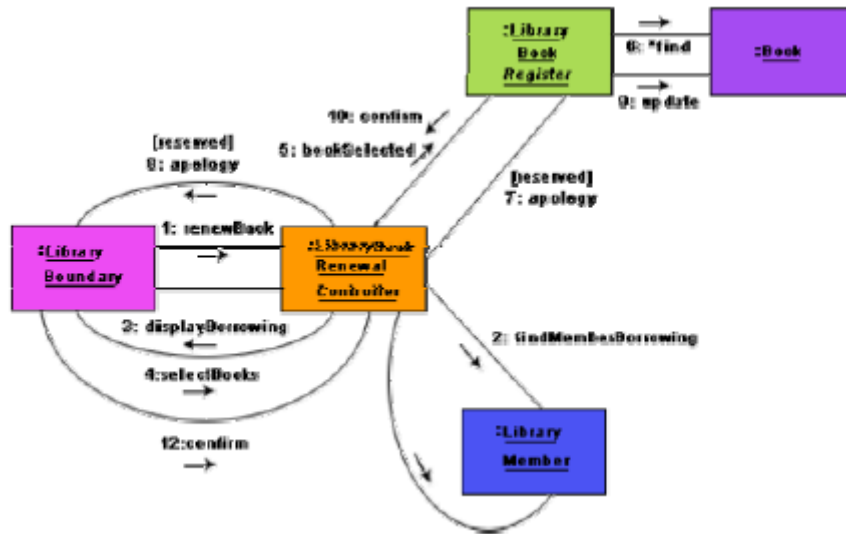


Fig. 7.14: Collaboration diagram for the renew book use cases

7.4 ACTIVITY AND STATE CHART DIAGRAM

7.4.1 Activity diagrams

The activity diagram is possibly one modeling element which was not present in any of the predecessors of UML. No such diagrams were present either in the works of Booch, Jacobson, or Rumbaugh. It is possibly based on the event diagram of Odell [1992] through the notation is very different from that used by Odell. The activity diagram focuses on representing activities or chunks of processing which may or may not correspond to the methods of classes. An activity is a state with an internal action and one or more outgoing transitions which automatically follow the termination of the internal activity. If an activity has more than one outgoing transitions, then these must be identified through conditions. An interesting feature of the activity diagrams is the swim lanes. Swim lanes enable you to group activities based on who is performing them, e.g. academic department vs. hostel office. Thus swim lanes subdivide activities based on the responsibilities of some components. The activities in a swim lane can be assigned to some model elements, e.g. classes or some component, etc.

Activity diagrams are normally employed in business process modeling. This is carried out during the initial stages of requirements analysis and specification. Activity diagrams can be very useful to understand complex processing activities involving many components. Later these diagrams can be used to develop interaction diagrams which help to allocate activities (responsibilities) to classes. The student admission process in IIT is shown as an activity diagram in fig. 7.15. This shows the part played by different components of the Institute in the admission procedure. After the fees are received at the account section, parallel activities start at the hostel office, hospital, and the Department. After all these activities complete (this synchronization is represented as a horizontal line), the identity card can be issued to a student by the Academic section.

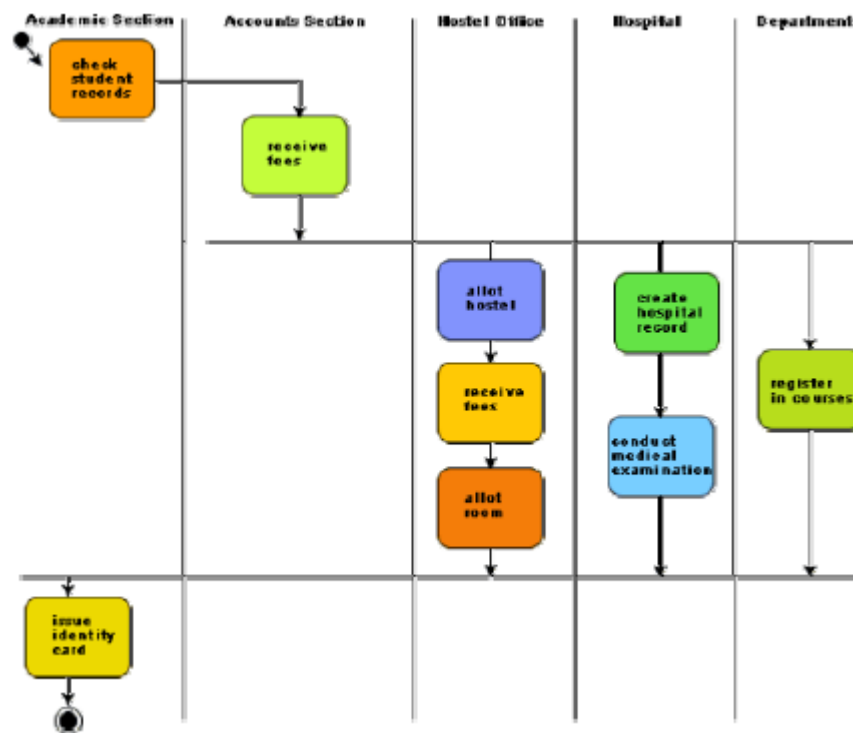


Fig. 7.15: Activity diagram for student admission procedure at IIT

7.4.2 Activity diagrams vs. procedural flow charts

Activity diagrams are similar to the procedural flow charts. The difference is that activity diagrams support description of parallel activities and synchronization aspects involved in different activities.

7.4.3 State chart diagram

A state chart diagram is normally used to model how the state of an object changes in its lifetime. State chart diagrams are good at describing how the behavior of an object changes across several use case executions. However, if we are interested in modeling some behavior that involves several objects collaborating with each other, state chart diagram is not appropriate. State chart diagrams are based on the finite state machine (FSM) formalism.

An FSM consists of a finite number of states corresponding to those of the object being modeled. The object undergoes state changes when specific events occur. The FSM formalism existed long before the object-oriented technology and has been used for a wide variety of applications. Apart from modeling, it has even been used in theoretical computer science as a generator for regular languages.

A major disadvantage of the FSM formalism is the state explosion problem. The number of states becomes too many and the model too complex when used to model practical systems. This problem is overcome in UML by using state charts. The state chart formalism was proposed by David Harel [1990]. A state chart is a hierarchical model of a system and introduces the concept of a composite state (also called nested state).

Actions are associated with transitions and are considered to be processes that occur quickly and are not interruptible. Activities are associated with states and can take longer. An activity can be interrupted by an event.

The basic elements of the state chart diagram are as follows:

- **Initial state.** This is represented as a filled circle.
- **Final state.** This is represented by a filled circle inside a larger circle.
- **State.** These are represented by rectangles with rounded corners.
- **Transition.** A transition is shown as an arrow between two states. Normally, the name of the event which causes the transition is placed along side the arrow. A guard to the transition can also be assigned. A guard is a Boolean logic condition. The transition can take place only if the guard evaluates to true. The syntax for the label of the transition is shown in 3 parts: event[guard]/action.

An example state chart for the order object of the Trade House Automation software is shown in fig. 7.16.



Fig. 7.16: State chart diagram for an order object

7.4.4 Activity diagram vs. State chart diagram

- Both activity and state chart diagrams model the dynamic behavior of the system. Activity diagram is essentially a flowchart showing flow of control from activity to activity. A state chart diagram shows a state machine emphasizing the flow of control from state to state.
- An activity diagram is a special case of a state chart diagram in which all or most of the states are activity states and all or most of the transitions are triggered by completion of activities in the source state (An activity is an ongoing non-atomic execution within a state machine).
- Activity diagrams may stand alone to visualize, specify, and document the dynamics of a society of objects or they may be used to model the flow of control of an operation. State chart diagrams may be attached to classes, use cases, or entire systems in order to visualize, specify, and document the dynamics of an individual object.

Check Your Progress II

1. In the context of use case diagram, the stick person icon is used to represent
 - a. human users ✓
 - b. external systems
 - c. internal systems
 - d. none of the above

State True or False

2. In a UML class diagram, the aggregation relationship defines an equivalence relationship among objects.
3. The aggregation relationship can be considered to be a special type of association relationship.
4. The aggregation relationship can be reflexive.
5. The aggregation relationship cannot be reflexive and symmetric but is transitive.
6. Normally, you use an interaction diagram to represent how the behavior of an object changes over its life time.
7. The chronological order of the messages in an interaction diagram cannot be determined from an inspection of the diagram.
8. The interaction diagrams can be effectively used to describe how the behavior of an object changes across several use case.
9. State chart diagrams in UML are normally used to model how some behavior of a system is realized through the co-operative actions of several objects.
10. A state chart diagram is good at describing behavior that involves multiple objects cooperating with each other to achieve some behavior.

7.5 SUMMARY

1. For a system in which the objects undergo many state changes, a state chart diagram may be necessary. For a system, which is implemented on a large number of hardware components, a deployment diagram may be

necessary. So, the type of models to be constructed depends on the problem at hand.

2. An important reason behind constructing a model is that it helps manage complexity. Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the Analysis, Specification, Code generation, Design, Visualize and understand the problem and the working of a system, Testing, etc.
3. A model that is being used for analysis and specification would not show any of the design decisions that would be made later on during the design stage.
4. UML can be used to construct nine different types of diagrams to capture five different views of a system.
5. Different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system.
6. The UML diagrams can capture the following five views of a system: User's view, Structural view, Behavioral view, Implementation view and Environmental view.
7. User's view defines the functionalities (facilities) made available by the system to its users.
8. The users' view captures the external users' view of the system in terms of the functionalities offered by the system.
9. The users' view is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible.
10. The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects).
11. The structural model is also called the static model, since the structure of a system does not change with time.

12. The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.
13. Implementation view captures the important components of the system and their dependencies.
14. Environmental view models how the different components are implemented on different pieces of hardware.
15. The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the system.
16. The use cases do not mention any specific algorithm to be used or the internal data representation, internal structure of the software, etc.
17. A use case typically represents a sequence of interactions between the user and the system. These interactions consist of one mainline sequence.
18. Implementation view captures the important components of the system and their dependencies.

7.6 MODEL QUESTIONS

1. Explain why is it necessary to create a model in the context of good software development.
2. Identify different types of views of a system captured by UML diagrams.
3. What is the purpose of a use case?
4. Which diagrams in UML capture the behavioral view of the system?
5. Which UML diagrams capture the structural aspects of a system?
6. Which UML diagrams capture the important components of the system and their dependencies?
7. Represent the following relations among classes using UML diagram.
8. What is the necessity for developing use case diagram?
9. How to identify use cases of a system?
10. What is the difference between an operation and a method in the context of object-oriented design technique?
11. What does the association relationship among classes represent? Give examples of the association relationship.

12. What does aggregation relationship between classes represent? Give examples of aggregation relationship between classes.
13. Why are objects always passed by reference in all popular programming languages?

Answers to Check Your Progress I

1. d
2. c
3. False
4. False
5. False
6. False
7. False
8. True
9. False

Answers to Check Your Progress II

1. a
2. False
3. True
4. False
5. True
6. False
7. False
8. False
9. False
10. False

UNIT VIII: OBJECT-ORIENTED SOFTWARE DEVELOPMENT

8.0 LEARNING OBJECTIVES

At the end of this lesson the student will be able to:

- Identify the basic difference between object-oriented analysis (OOA) and object-oriented design (OOD).
- Explain why design patterns are important in creating good software design.
- Explain what are design patterns.
- Identify pattern solution for a particular problem in terms of class and interaction diagrams.
- Explain expert pattern and circumstances when it can be used.
- Explain creator pattern and circumstances when it can be used.
- Explain controller pattern and circumstances when it can be used.
- Explain facade pattern and circumstances when it can be used.
- Explain model view separation pattern and circumstances when it can be used.
- Explain intermediary pattern (i.e. proxy pattern) and circumstances when it can be used.
- Explain what is meant by domain modeling.
- Identify the three types of objects identified during domain analysis.
- Explain the purpose of different types of objects identified during domain analysis. Explain how these objects interact among each other.
- Explain at least three approaches for identifying objects in the context of object-oriented design methodology.
- Identify two goals of interaction modeling.
- Explain the CRC cards technique.
- Develop sequence diagram for any given use case.

- Identify how sequence diagrams are useful in developing the class diagram.
- Identify five important criteria for judging the goodness of an object-oriented design.

8.1 DESIGN PATTERNS

The term object-oriented analysis (OOA) refers to a method of developing an initial model of the software from the requirements specification. The analysis model is refined into a design model. The design model can be implemented using a programming language. The term object-oriented programming refers to the implementation of programs using object-oriented concepts.

In contrast, object-oriented design (OOD) paradigm suggests that the natural objects (i.e. the entities) occurring in a problem should be identified first and then implemented. Object-oriented design (OOD) techniques not only identify objects but also identify the internal details of these identified objects. Also, the relationships existing among different objects are identified and represented in such a way that the objects can be easily implemented using a programming language.

Design patterns are reusable solutions to problems that recur in many applications. A pattern serves as a guide for creating a “good” design. Patterns are based on sound common sense and the application of fundamental design principles. These are created by people who spot repeating themes across designs. The pattern solutions are typically described in terms of class and interaction diagrams. Examples of design patterns are expert pattern, creator pattern, controller pattern etc.

Design patterns are very useful in creating good software design solutions. In addition to providing the model of a good solution, design patterns include a clear specification of the problem, and also explain the circumstances in which the solution would and would not work. Thus, a design pattern has four important parts:

- The problem.
- The context in which the problem occurs.
- The solution.

- The context within which the solution works.

The design pattern solutions are typically described in terms of class and interaction diagrams.

Example:

Expert Pattern

Problem: Which class should be responsible for doing certain things?

Solution: Assign responsibility to the information expert – the class that has the information necessary to fulfill the required responsibility. The expert pattern expresses the common intuition that objects do things related to the information they have. The class diagram and collaboration diagrams for this solution to the problem of which class should compute the total sales is shown in the fig. 8.1.

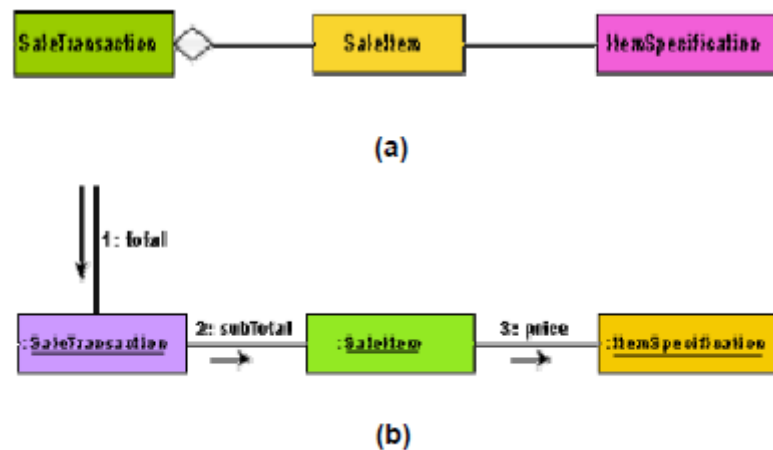


Fig. 8.1: Expert pattern: (a) Class diagram (b) Collaboration diagram

Creator Pattern

Problem: Which class should be responsible for creating a new instance of some class?

Solution: Assign a class C1 the responsibility to create an instance of class C2, if one or more of the following are true:

- C1 is an aggregation of objects of type C2.
- C1 contains objects of type C2.
- C1 closely uses objects of type C2.
- C1 has the data that would be required to initialize the objects of type C2, when they are created.

Controller Pattern:

Problem: Who should be responsible for handling the actor requests?

Solution: For every use case, there should be a separate controller object which would be responsible for handling requests from the actor. Also, the same controller should be used for all the actor requests pertaining to one use case so that it becomes possible to maintain the necessary information about the state of the use case. The state information maintained by a controller can be used to identify the out-of-sequence actor requests, e.g. whether voucher request is received before arrange payment request.

Façade Pattern:

Problem: How should the services be requested from a service package?

Context in which the problem occurs: A package as already discussed is a cohesive set of classes – the classes have strongly related responsibilities. For example, an RDBMS interface package may contain classes that allow one to perform various operations on the RDBMS.

Solution: A class (such as DBfacade) can be created which provides a common interface to the services of the package.

Model View Separation Pattern:

Problem: How should the non-GUI classes communicate with the GUI classes?

Context in which the problem occurs: This is a very commonly occurring pattern which occurs in almost every problem. Here, model is a synonym for the domain layer objects, view is a synonym for the presentation layer objects such as the GUI objects.

Solution: The model view separation pattern states that model objects should not have direct knowledge (or be directly coupled) to the view objects. This means that there should not be any direct calls from other objects to the GUI objects. This results in a good solution, because the GUI classes are related to a particular application whereas the other classes may be reused.

There are actually two solutions to this problem which work in different circumstances as follows:

Solution 1: Polling or Pull from above

It is the responsibility of a GUI object to ask for the relevant information from the other objects, i.e. the GUI objects pull the necessary information from the other objects whenever required.

This model is frequently used. However, it is inefficient for certain applications. For example, simulation applications which require visualization, the GUI objects would not know when the necessary information becomes available. Other examples are, monitoring applications such as network monitoring, stock market quotes, and so on. In these situations, a “push-from-below” model of display update is required. Since “push-from-below” is not an acceptable solution, an indirect mode of communication from the other objects to the GUI objects is required.

Solution 2: Publish- subscribe pattern

An event notification system is implemented through which the publisher can indirectly notify the subscribers as soon as the necessary information becomes available. An event manager class can be defined which keeps track of the subscribers and the types of events they are interested in. An event is published by the publisher by sending a message to the event manager object. The event manager notifies all registered subscribers usually via a parameterized message (called a callback). Some languages specifically support event manager classes. For example, Java provides the Event Listener interface for such purposes.

Intermediary Pattern or Proxy

Problem: How should the client and server objects interact with each other?

Context in the problem occurs: The client and server terms as used here refer to software components existing across a network. The clients are consumers of services provided by the servers.

Solution: A proxy object at the client side can be defined which is a local sit-in for the remote server object. The proxy hides the details of the network transmission. The proxy is responsible for determining the server address, communicating the client request to the server, obtaining the server response and seamlessly passing that to the client. The proxy can also augment (or filter) information that is exchanged between the client and the server. The proxy could have the same interface as the remote server object so that the client feels as if it is interacting directly with the remote server object and the complexities of network transmissions are abstracted out.

Check Your Progress I

Mark all options which are true.

1. The design pattern solutions are typically described in terms of
 - a. class diagrams
 - b. object diagrams
 - c. interaction diagrams
 - d. both class and interaction diagrams
2. The class that should be responsible for doing certain things for which it has the necessary information – is the solution proposed by
 - a. creator pattern
 - b. controller pattern
 - c. expert pattern
 - d. façade pattern
3. The class that should be responsible for creating a new instance of some class – is the solution proposed by
 - a. creator pattern
 - b. controller pattern
 - c. expert pattern
 - d. façade pattern
4. The class that should be responsible for handling the actor requests – is the solution proposed by
 - a. creator pattern
 - b. controller pattern
 - c. expert pattern
 - d. façade pattern

True or False

5. Façade pattern tells the way that non-GUI classes should communicate with the GUI classes.
6. The use cases should be tightly tied to the GUI.
7. The responsibilities assigned to a controller object are closely related to the realization of a specific use case.
8. Entity objects are responsible for implementing the business logic.

8.2 DOMAIN MODELING

Domain modeling is known as conceptual modeling. A domain model is a representation of the concepts or objects appearing in the problem domain. It also captures the obvious relationships among these objects. Examples of such conceptual objects are the Book, BookRegister, MemeberRegister, LibraryMember, etc. The recommended strategy is to quickly create a rough conceptual model where the emphasis is in finding the obvious concepts expressed in the requirements while deferring a detailed investigation. Later during the development process, the conceptual model is incrementally refined and extended. Identify the three types of objects identified during domain analysis.

The objects identified during domain analysis can be classified into three types:

- Boundary objects
- Controller objects
- Entity objects

The boundary and controller objects can be systematically identified from the use case diagram whereas identification of entity objects requires practice. So, the crux of the domain modeling activity is to identify the entity models. The different kinds of objects identified during domain analysis and their relationships are as follows:

Boundary objects: The boundary objects are those with which the actors interact. These include screens, menus, forms, dialogs, etc. The boundary objects are mainly responsible for user interaction. Therefore, they normally do not include any processing logic. However, they may be responsible for validating inputs, formatting, outputs, etc. The boundary objects were earlier being called as the interface objects. However, the term interface class is being used for Java, COM/DCOM, and UML with different meaning. A recommendation for the initial identification of the boundary classes is to define one boundary class per actor/use case pair.

Entity objects: These normally hold information such as data tables and files that need to outlive use case execution, e.g. Book, BookRegister, LibraryMember, etc. Many of the entity objects are “dumb servers”. They are normally responsible for

storing data, fetching data, and doing some fundamental kinds of operation that do not change often.

Controller objects: The controller objects coordinate the activities of a set of entity objects and interface with the boundary objects to provide the overall behavior of the system. The responsibilities assigned to a controller object are closely related to the realization of a specific use case. The controller objects effectively decouple the boundary and entity objects from one another making the system tolerant to changes of the user interface and processing logic. The controller objects embody most of the logic involved with the use case realization (this logic may change time to time). A typical interaction of a controller object with boundary and entity objects is shown in fig. 8.2. Normally, each use case is realized using one controller object. However, some use cases can be realized without using any controller object, i.e. through boundary and entity objects only. This is often true for use cases that achieve only some simple manipulation of the stored information.

For example, let's consider the "query book availability" use case of the Library Information System (LIS). Realization of the use case involves only matching the given book name against the books available in the catalog. More complex use cases may require more than one controller object to realize the use case. A complex use case can have several controller objects such as transaction manager, resource coordinator, and error handler. There is another situation where a use case can have more than one controller object. Sometimes the use cases require the controller object to transit through a number of states. In such cases, one controller object might have to be created for each execution of the use case.

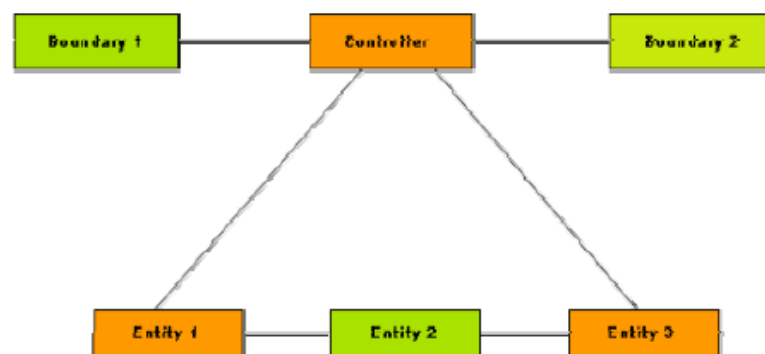


Fig. 8.2: A typical realization of a use case through the collaboration of boundary, controller, and entity objects

One of the most important steps in any object-oriented design methodology is the identification of objects. In fact, the quality of the final design depends to a great extent on the appropriateness of the objects identified. However, to date no formal methodology exists for identification of objects. Several semi-formal and informal approaches have been proposed for object identification. These can be classified into the following broad classes:

- Grammatical analysis of the problem description.
- Derivation from data flow.
- Derivation from the entity relationship (E-R) diagram.

A widely accepted object identification approach is the grammatical analysis approach. Grady Booch originated the grammatical analysis approach [1991]. In Booch's approach, the nouns occurring in the extended problem description statement (processing narrative) are mapped to objects and the verbs are mapped to methods. The identification approaches based on derivation from the data flow diagram and the entity-relationship model are still evolving and therefore will not be discussed in this text.

8.2.1 Booch's Object Identification Method

Booch's object identification approach requires a processing narrative of the given problem to be first developed. The processing narrative describes the problem and discusses how it can be solved. The objects are identified by noting down the nouns in the processing narrative. Synonym of a noun must be eliminated. If an object is required to implement a solution, then it is said to be part of the solution space. Otherwise, if an object is necessary only to describe the problem, then it is said to be a part of the problem space. However, several of the nouns may not be objects. An imperative procedure name, i.e., noun form of a verb actually represents an action and should not be considered as an object. A potential object found after lexical analysis is usually considered legitimate, only if it satisfies the following criteria:

Retained information. Some information about the object should be remembered for the system to function. If an object does not contain any private data, it can not be expected to play any important role in the system.

Multiple attributes. Usually objects have multiple attributes and support multiple methods. It is very rare to find useful objects which store only a single data element or support only a single method, because an object having only a single data element or method is usually implemented as a part of another object.

Common operations. A set of operations can be defined for potential objects. If these operations apply to all occurrences of the object, then a class can be defined. An attribute or operation defined for a class must apply to each instance of the class. If some of the attributes or operations apply only to some specific instances of the class, then one or more subclasses can be needed for these special objects.

Normally, the actors themselves and the interactions among themselves should be excluded from the entity identification exercise. However, some times there is a need to maintain information about an actor within the system. This is not the same as modeling the actor. These classes are sometimes called surrogates. For example, in the Library Information System (LIS) we would need to store information about each library member. This is independent of the fact that the library member also plays the role of an actor of the system.

Although the grammatical approach is simple and intuitively appealing, yet through a naive use of the approach, it is very difficult to achieve high quality results. In particular, it is very difficult to come up with useful abstractions simply by doing grammatical analysis of the problem description. Useful abstractions usually result from clever factoring of the problem description into independent and intuitively correct elements.

Example: Tic-tac-toe

Let us identify the entity objects of the following Tic-tac-toe software:

Tic-tac-toe is a computer game in which a human player and the computer **make** alternative moves on a 3 X 3 square. A move consists of **marking** a previously unmarked square. A player who first **places** three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square **wins**. As soon as either the human player or the computer **wins**, a message congratulating the winner should be **displayed**. If neither player manages to **get** three consecutive

marks along a straight line, and all the squares on the board are filled up, then the game is **drawn**. The computer always **tries to win** a game.

By performing a grammatical analysis of this problem statement, it can be seen that nouns that have been underlined in the problem description and the actions as the italicized verbs. However, on closer examination synonyms can be eliminated from the identified nouns. The list of nouns after eliminating the synonyms are the following: Tic-tac-toe, computer game, human player, move, square, mark, straight line, board, row, column, and diagonal.

From this list of possible objects, nouns can be eliminated like human player as it does not belong to the problem domain. Also, the nouns square, game, computer, Tic-tac-toe, straight line, row, column, and diagonal can be eliminated, as any data and methods can not be associated with them. The noun **move** can also be eliminated from the list of potential objects since it is an imperative verb and actually represents an action. Thus, there is only one object left – board.

After experienced in object identification, it is not normally necessary to really identify all nouns in the problem description by underlining them or actually listing them down, and systematically eliminate the non-objects to arrive at the final set of objects. The primary goals of interaction modeling are the following:

- To allocate the responsibility of a use case realization among the boundary, entity, and controller objects. The responsibilities for each class are reflected as an operation to be supported by that class.
- To show the detailed interaction that occurs over time among the objects associated with each use case.

The interactions diagrams for only simple use cases that involve collaboration among a limited number of classes can be drawn from an inspection of the use case description. More complex use cases require the use of CRC cards where a number of team members participate to determine the responsibility of the classes involved in the use case realization.

CRC (Class-Responsibility-Collaborator) technology was pioneered by Ward Cunningham and Kent Becka at the research laboratory of Tektronix at Portland, Oregon, USA. CRC cards are index cards that are prepared one per each class. On each of these cards, the responsibility of each class is written briefly. The objects with which this object needs to collaborate its responsibility are also written.

CRC cards are usually developed in small group sessions where people role play being various classes. Each person holds the CRC card of the classes he is playing the role of. The cards are deliberately made small (4 inch ´ 6 inch) so that each class can have only limited number of responsibilities. A responsibility is the high level description of the part that a class needs to play in the realization of a use case. An example CRC card for the BookRegister class of the Library Automation System is shown in fig. 8.3.

After assigning the responsibility to classes using CRC cards, it is easier to develop the interaction diagrams by flipping through the CRC cards.

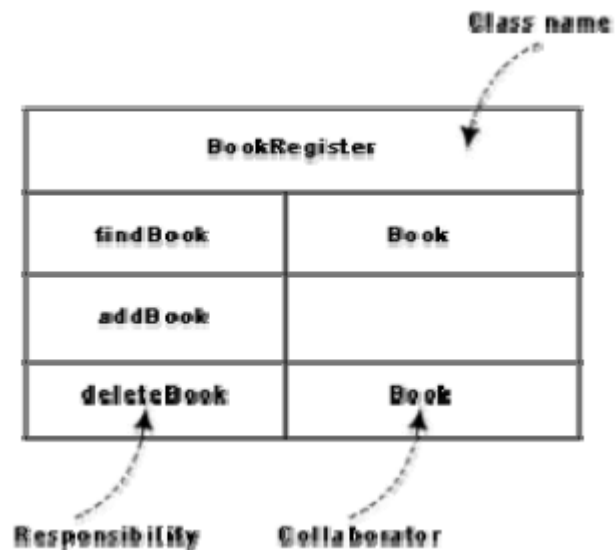


Fig. 8.3: CRC card for the Book Register class

Consider the Tic-tac-toe computer game discussed earlier. The step-by-step workout of this example is as follows:

- The use case model is shown in fig. 7.2.
- The initial domain model is shown in fig. 8.4(a).
- The domain model after adding the boundary and control classes is shown in fig. 8.4(b).
- Sequence diagram for the play move use case is shown in fig. 8.5.

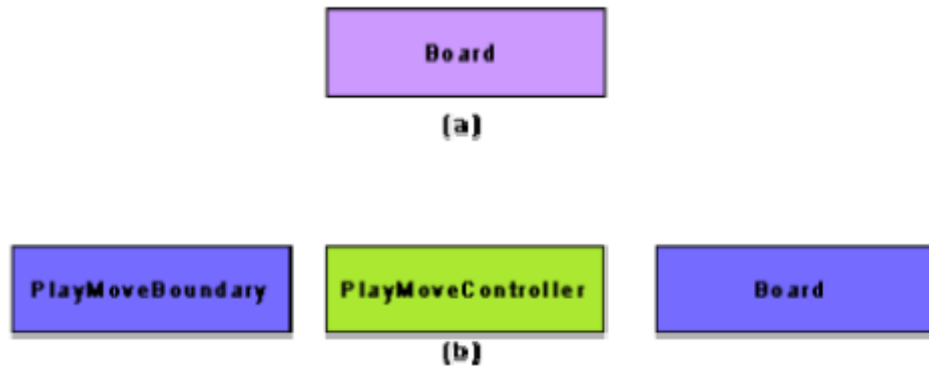


Fig. 8.4: (a) Initial domain model (b) Refined domain model

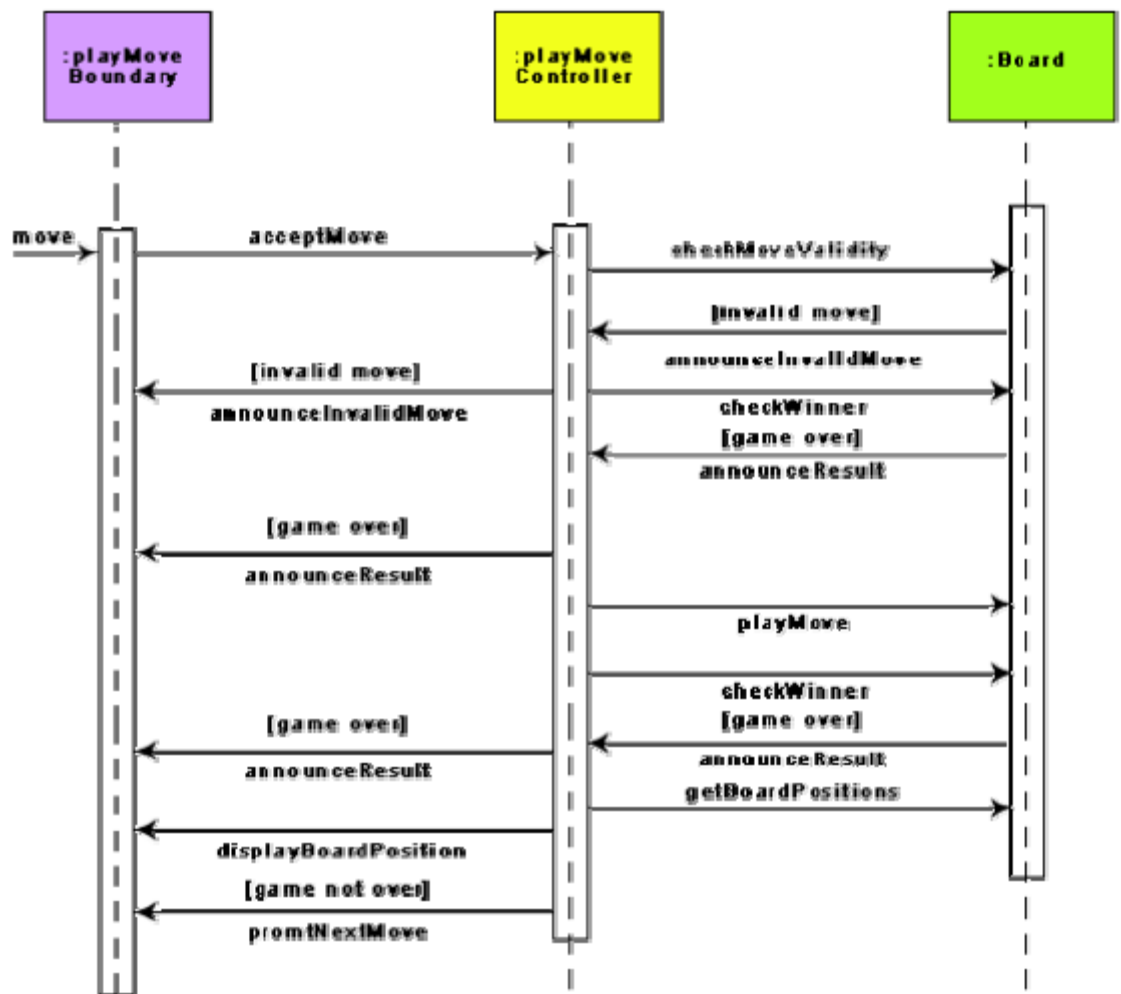


Fig. 8.5: Sequence diagram for the play move use case

Consider the Supermarket prizes scheme software discussed earlier. The step-by-step analysis and design workout of this problem is as follows:

- The use case model is shown in fig. 8.6.
- The initial domain model is shown in fig. 8.7(a).
- The domain model after adding the boundary and control classes is shown in fig. 8.7(b).
- Sequence diagram for the select winner list use case is shown in fig. 8.8.
- Sequence diagram for the register customer use case is shown in fig. 8.9.
- Sequence diagram for the register sales use case is shown in fig. 8.10. In this use case, since the responsibility of the RegisterSalesController is trivial, the controller class can be deleted and the sequence diagram can be redrawn as in fig. 8.11 after incorporating this change.
- Class diagram is shown in fig. 8.12. The messages of the sequence diagrams of the different use cases have been populated as the methods of the corresponding classes.

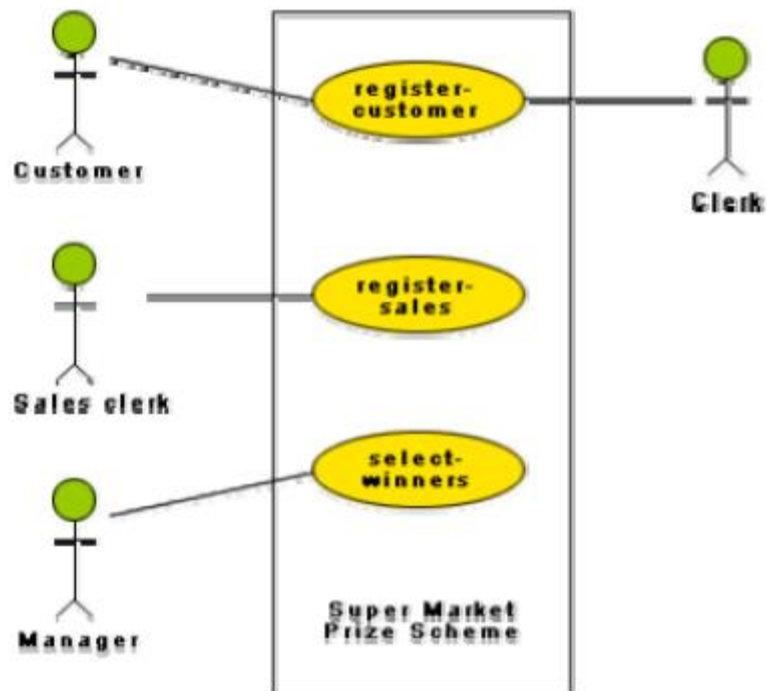
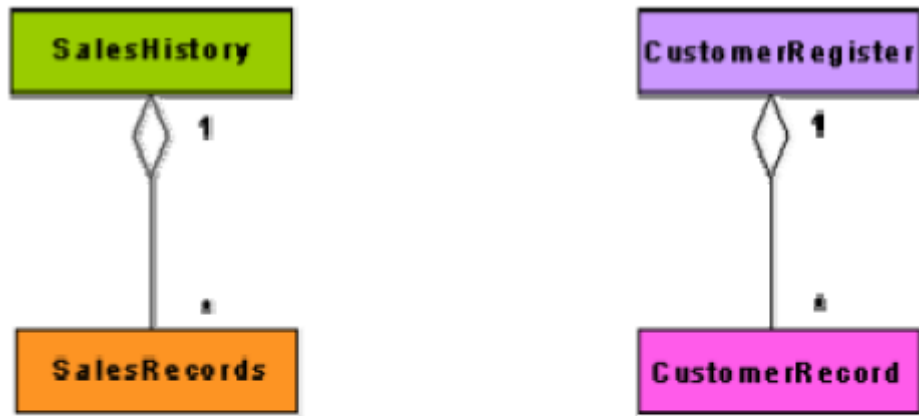
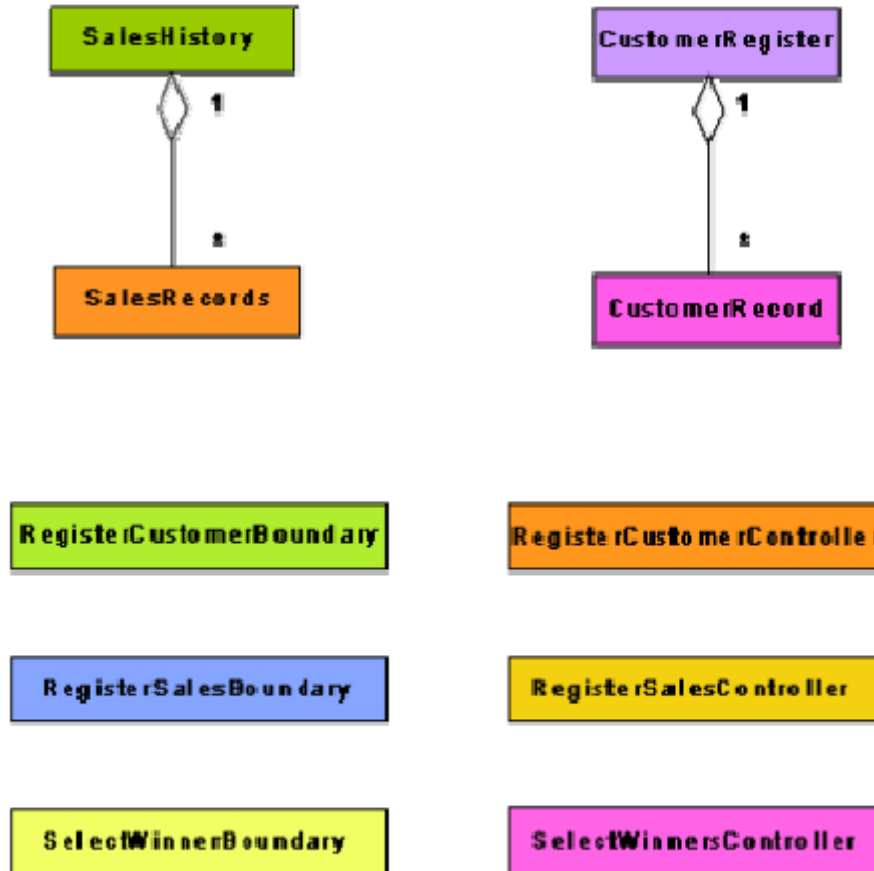


Fig. 8.6: Use case model for Super Market Prize Scheme



(a)



(b)

Fig. 8.7: (a) Initial domain model (b) Refined domain model

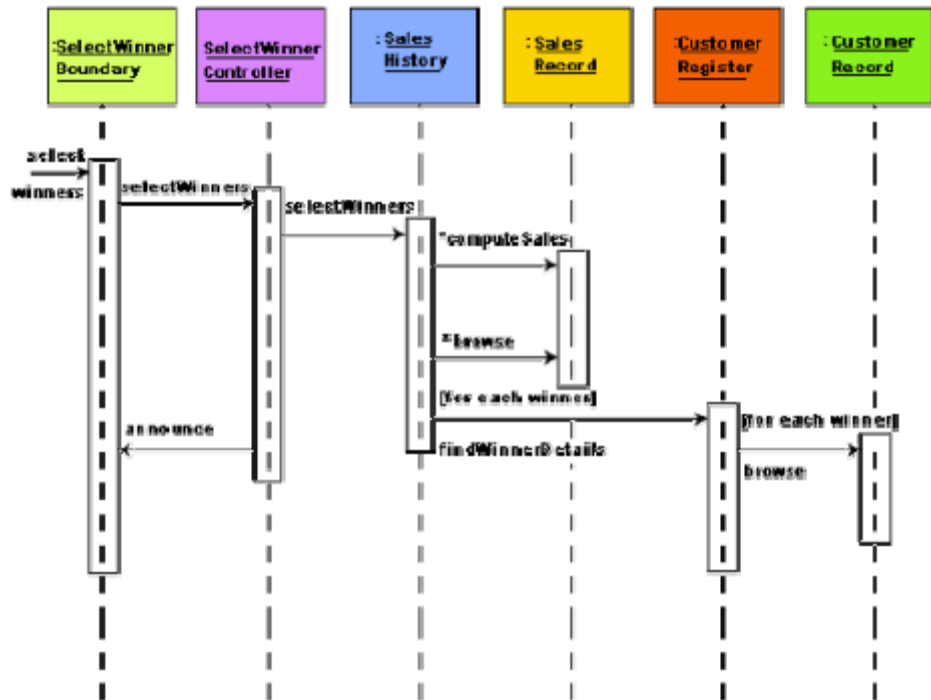


Fig. 8.8: Sequence diagram for the select winner list use case

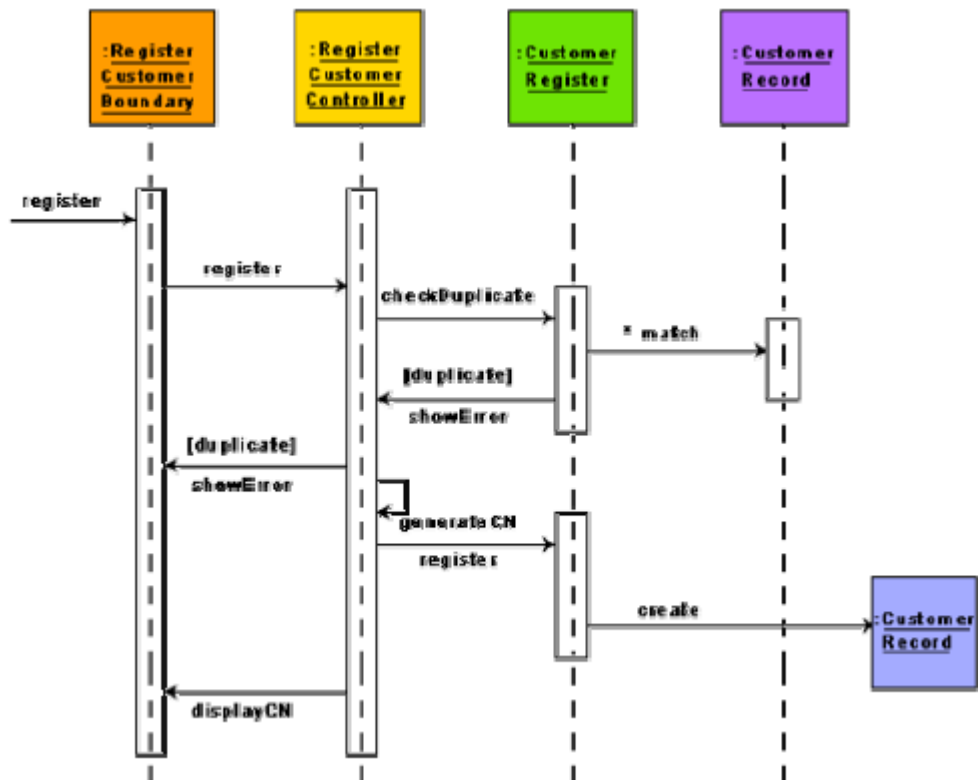


Fig. 8.9: Sequence diagram for the register customer use case

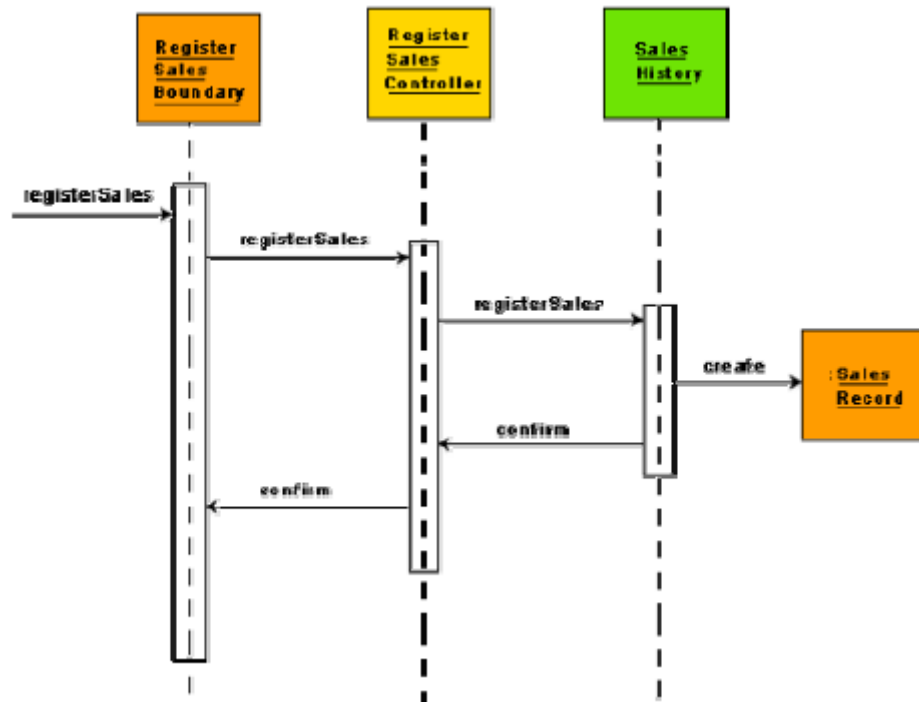


Fig. 8.10: Sequence diagram for the register sales use case

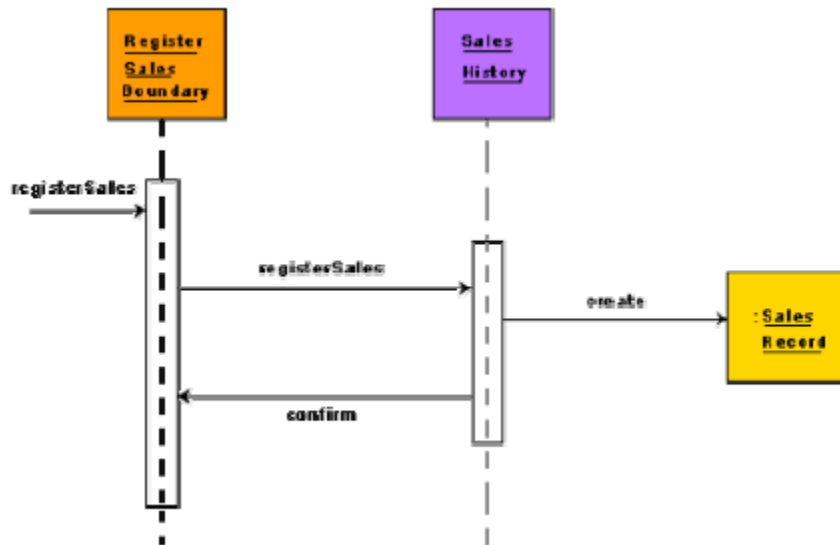


Fig. 8.11: Refined sequence diagram for the register sales use case

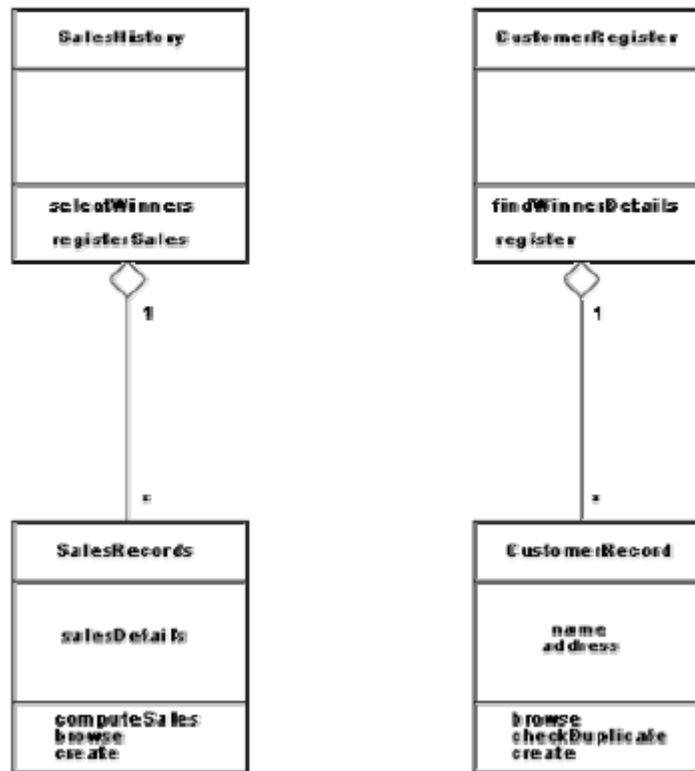


Fig. 8.12: Class diagram

It is quite obvious that there are several subjective judgments involved in arriving at a good object-oriented design. Therefore, several alternative design solutions to the same problem are possible. In order to be able to determine which of any two designs is better, some criteria for judging the goodness of a design must be identified. The following are some of the accepted criteria for judging the goodness of a design.

- **Coupling guidelines.** The number of messages between two objects or among a group of objects should be minimum. Excessive coupling between objects is detrimental to modular design and prevents reuse.
- **Cohesion guideline.** In OOD, cohesion is about three levels: # Cohesiveness of the individual methods. Cohesiveness of each of the individual method is desirable, since it assumes that each method does only a well-defined function. # Cohesiveness of the data and methods within a class. This is desirable since it assures that the methods of an object do actions for which the object is naturally responsible, i.e. it assures that no action has been improperly mapped to an object. #

Cohesiveness of an entire class hierarchy. Cohesiveness of methods within a class is desirable since it promotes encapsulation of the objects.

- **Hierarchy and factoring guidelines.** A base class should not have too many subclasses. If too many subclasses are derived from a single base class, then it becomes difficult to understand the design. In fact, there should approximately be no more than 7 ± 2 classes derived from a base class at any level.
- **Keeping message protocols simple.** Complex message protocols are an indication of excessive coupling among objects. If a message requires more than 3 parameters, then it is an indication of bad design.
- **Number of Methods.** Objects with a large number of methods are likely to be more application-specific and also difficult to comprehend – limiting the possibility of their reuse. Therefore, objects should not have too many methods. This is a measure of the complexity of a class. It is likely that the classes having more than about seven methods would have problems.
- **Depth of the inheritance tree.** The deeper a class is in the class inheritance hierarchy, the greater is the number of methods it is likely to inherit, making it more complex. Therefore, the height of the inheritance tree should not be very large.
- **Number of messages per use case.** If methods of a large number of objects are invoked in a chain action in response to a single message, testing and debugging of the objects becomes complicated. Therefore, a single message should not result in excessive message generation and transmission in a system.
- **Response for a class.** This is a measure of the maximum number of methods that an instance of this class would call. If the same method is called more than once, then it is counted only once. A class which calls more than about seven different methods is susceptible to errors.

Check Your Progress II

Mark all options which are true.

1. The objects identified during domain analysis can be classified into
 - a. boundary objects

- b. controller objects
 - c. entity objects
 - d. all of the above
2. The objects with which the actors interact are known as
- a. controller objects
 - b. boundary objects
 - c. entity objects
 - d. all of the above
3. The most critical part of the domain modeling activity is to identify
- a. controller objects
 - b. boundary objects
 - c. entity objects
 - d. none of the above
4. The objects which are responsible for storing data, fetching data, and doing some fundamental kinds of operation that do not change often are known as
- a. controller objects
 - b. boundary objects
 - c. entity objects
 - d. none of the above
5. The objects which effectively decouple the boundary and entity objects from one another making the system tolerant to changes of the user interface and processing logic are
- a. controller objects
 - b. boundary objects
 - c. entity objects
 - d. none of the above

Mark the following as either True or False.

6. CRC card technique is useful in identifying the different classes necessary to solve a problem.
7. There is a one-to-one correspondence between the classes of the domain model and the final class diagram.

8. Large number of message exchanges between objects indicates good delegation and is a sure sign of a design well-done.
9. Deep class hierarchies are the hallmark of any good OOD.
10. Cohesiveness of the data and methods within a class is a sign of good OOD.
11. Keeping the message protocols complex is an indication of a good object-oriented design.

8.3 SUMMARY

1. The term object-oriented analysis (OOA) refers to a method of developing an initial model of the software from the requirements specification.
2. The analysis model is refined into a design model. The design model can be implemented using a programming language.
3. The term object-oriented programming refers to the implementation of programs using object-oriented concepts.
4. In contrast, object-oriented design (OOD) paradigm suggests that the natural objects (i.e. the entities) occurring in a problem should be identified first and then implemented. Object-oriented design (OOD) techniques not only identify objects but also identify the internal details of these identified objects. Also, the relationships existing among different objects are identified and represented in such a way that the objects can be easily implemented using a programming language.
5. Design patterns are very useful in creating good software design solutions. In addition to providing the model of a good solution, design patterns include a clear specification of the problem, and also explain the circumstances in which the solution would and would not work.
6. Design patterns are reusable solutions to problems that recur in many applications.
7. Patterns are based on sound common sense and the application of fundamental design principles. These are created by people who spot repeating themes across designs.

8. The pattern solutions are typically described in terms of class and interaction diagrams. Examples of design patterns are expert pattern, creator pattern, controller pattern etc.
9. Domain modeling is known as conceptual modeling. A domain model is a representation of the concepts or objects appearing in the problem domain. It also captures the obvious relationships among these objects. Examples of such conceptual objects are the Book, BookRegister, MemeberRegister, LibraryMember, etc. The recommended strategy is to quickly create a rough conceptual model where the emphasis is in finding the obvious concepts expressed in the requirements while deferring a detailed investigation. Later during the development process, the conceptual model is incrementally refined and extended.
10. Booch's object identification approach requires a processing narrative of the given problem to be first developed.
11. The processing narrative describes the problem and discusses how it can be solved. The objects are identified by noting down the nouns in the processing narrative. Synonym of a noun must be eliminated.
12. If an object is required to implement a solution, then it is said to be part of the solution space. Otherwise, if an object is necessary only to describe the problem, then it is said to be a part of the problem space. However, several of the nouns may not be objects.
13. An imperative procedure name, i.e., noun form of a verb actually represents an action and should not be considered as an object.
14. The interactions diagrams for only simple use cases that involve collaboration among a limited number of classes can be drawn from an inspection of the use case description.
15. CRC (Class-Responsibility-Collaborator) technology was pioneered by Ward Cunningham and Kent Becka at the research laboratory of Tektronix at Portland, Oregon, USA.
16. CRC cards are index cards that are prepared one per each class. On each of these cards, the responsibility of each class is written briefly. The objects with which this object needs to collaborate its responsibility are also written.

17. CRC cards are usually developed in small group sessions where people role play being various classes. Each person holds the CRC card of the classes he is playing the role of. The cards are deliberately made small (4 inch ´ 6 inch) so that each class can have only limited number of responsibilities.
18. A responsibility is the high level description of the part that a class needs to play in the realization of a use case.
19. After assigning the responsibility to classes using CRC cards, it is easier to develop the interaction diagrams by flipping through the CRC cards.
20. Cohesion is a measure of functional strength of a module.

8.4 MODEL QUESTIONS

1. Write down basic differences between object-oriented analysis (OOA) and object-oriented design (OOD) technique.
2. What is meant by design patterns?
3. What are the advantages of using design patterns?
4. Write down some popular design patterns and their necessities.
5. Outline an object-oriented development process.
6. What is meant by domain modeling?
7. Differentiate among different types of objects that are identified during domain analysis and also explain the relationships among those identified objects.
8. Explain at least three approaches for identifying objects in the context of object-oriented design methodology.
9. Identify the primary goal of interaction modeling in the context of object-oriented design.
10. Identify the necessity of CRC (Class-Responsibility-Collaborator) cards in the context of object-oriented design.
11. Define the term cohesion in the context of object-oriented design.
12. Identify at least five important features that characterize a good object-oriented design.

1.

Answers to Check Your Progress I

1. d
2. c
3. a
4. b
5. False
6. False
7. True
8. False

Answers to Check Your Progress II

1. d
2. b
3. c
4. c
5. a
6. False
7. False
8. False
9. False
10. True
11. False

UNIT IX: USER INTERFACE DESIGN

9.0 LEARNING OBJECTIVES

At the end of this lesson the student will be able to:

- Identify five desirable characteristics of a user interface.
- Differentiate between user guidance and online help system.
- Differentiate between a mode-based interface and the modeless interface.
- Compare various characteristics of a GUI with those of a text-based user interface.
- Classify user interfaces into three main types.
- What are the different ways in which menu items can be arranged when the menu choices are large.
- Identify three characteristics of command language-based interface.
- Identify the three disadvantages of command language-based interface.
- Identify three issues in designing a command language-based interface.
- Identify three main types of menus with their features.
- Explain what an iconic interface is.
- What is meant by component-based GUI development style.
- Explain the necessity of component-based GUI development.
- Explain what is window in terms of GUI.
- Explain what is meant by window management system.
- Identify the responsibilities of a window manager.
- Identify at least eight primary types of window objects.
- Explain what X Window is.
- Explain why the X Window is so popular.
- Explain architecture of an X System.
- Explain what is meant by visual programming.
- Differentiate between a user-centered design and a design by users.
- Explain implications of human cognition capabilities on user interface design.

- Define seven important steps needed to design a GUI methodology.
- Prepare a check list for user interface inspection.

9.1. BASIC CONCEPTS IN USER INTERFACE DESIGN

It is very important to identify the characteristics desired of a good user interface. Because unless we are aware of these, it is very much difficult to design a good user interface. A few important characteristics of a good user interface are the following:

- **Speed of learning.** A good user interface should be easy to learn. Speed of learning is hampered by complex syntax and semantics of the command issue procedures. A good user interface should not require its users to memorize commands. Neither should the user be asked to remember information from one screen to another while performing various tasks using the interface. Besides, the following three issues are crucial to enhance the speed of learning:
 - **Use of Metaphors and intuitive command names.** Speed of learning an interface is greatly facilitated if these are based on some day-to-day real-life examples or some physical objects with which the users are familiar. The abstractions of real-life objects or concepts used in user interface design are called metaphors. If the user interface of a text editor uses concepts similar to the tools used by a writer for text editing such as cutting lines and paragraphs and pasting it at other places, users can immediately relate to it. Another popular metaphor is a shopping cart. Everyone knows how a shopping cart is used to make choices while purchasing items in a supermarket. If a user interface uses the shopping cart metaphor for designing the interaction style for a situation where similar types of choices have to be made, then the users can easily understand and learn to use the interface. Yet another example of a metaphor is the trashcan. To delete a file, the user may drag it to the trashcan. Also, learning is facilitated by intuitive command names and symbolic command issue procedures.

- **Consistency.** Once a user learns about a command, he should be able to use the similar commands in different circumstances for carrying out similar actions. This makes it easier to learn the interface since the user can extend his knowledge about one part of the interface to the other parts. For example, in a word processor, “Control-b” is the short-cut key to embolden the selected text. The same short-cut should be used on the other parts of the interface, for example, to embolden text in graphic objects also - circle, rectangle, polygon, etc. Thus, the different commands supported by an interface should be consistent.
- **Component-based interface.** Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other applications with which the user is already familiar. This can be achieved if the interfaces of different applications are developed using some standard user interface components. This, in fact, is the theme of the component-based user interface. Examples of standard user interface components are: radio button, check box, text field, slider, progress bar, etc.

The speed of learning characteristic of a user interface can be determined by measuring the training time and practice that users require before they can effectively use the software.

- **Speed of use.** Speed of use of a user interface is determined by the time and user effort necessary to initiate and execute different commands. This characteristic of the interface is some times referred to as productivity support of the interface. It indicates how fast the users can perform their intended tasks. The time and user effort necessary to initiate and execute different commands should be minimal. This can be achieved through careful design of the interface. For example, an interface that requires users to type in lengthy commands or involves mouse movements to different areas of the screen that are wide apart for issuing commands can slow down the operating speed of users. The most frequently used commands should have the smallest length or be available at the top of the menu to minimize the mouse movements necessary to issue commands.

- **Speed of recall.** Once users learn how to use an interface, the speed with which they can recall the command issue procedure should be maximized. This characteristic is very important for intermittent users. Speed of recall is improved if the interface is based on some metaphors, symbolic command issue procedures, and intuitive command names.
- **Error prevention.** A good user interface should minimize the scope of committing errors while initiating different commands. The error rate of an interface can be easily determined by monitoring the errors committed by average users while using the interface. This monitoring can be automated by instrumenting the user interface code with monitoring code which can record the frequency and types of user error and later display the statistics of various kinds of errors committed by different users. Moreover, errors can be prevented by asking the users to confirm any potentially destructive actions specified by them, for example, deleting a group of files. Consistency of names, issue procedures, and behavior of similar commands and the simplicity of the command issue procedures minimize error possibilities. Also, the interface should prevent the user from entering wrong values.
- **Attractiveness.** A good user interface should be attractive to use. An attractive user interface catches user attention and fancy. In this respect, graphics-based user interfaces have a definite advantage over text-based interfaces.
- **Consistency.** The commands supported by a user interface should be consistent. The basic purpose of consistency is to allow users to generalize the knowledge about aspects of the interface from one part to another. Thus, consistency facilitates speed of learning, speed of recall, and also helps in reduction of error rate.
- **Feedback.** A good user interface must provide feedback to various user actions. Especially, if any user request takes more than few seconds to process, the user should be informed about the state of the processing of his request. In the absence of any response from the computer for a long time, a novice user might even start recovery/shutdown procedures in

panic. If required, the user should be periodically informed about the progress made in processing his command. For example, if the user specifies a file copy/file download operation, a progress bar can be displayed to display the status. This will help the user to monitor the status of the action initiated.

- **Support for multiple skill levels.** A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users. This is necessary because users with different levels of experience in using an application prefer different types of user interfaces. Experienced users are more concerned about the efficiency of the command issue procedure, whereas novice users pay importance to usability aspects. Very cryptic and complex commands discourage a novice, whereas elaborate command sequences make the command issue procedure very slow and therefore put off experienced users. When someone uses an application for the first time, his primary concern is speed of learning. After using an application for extended periods of time, he becomes familiar with the operation of the software. As a user becomes more and more familiar with an interface, his focus shifts from usability aspects to speed of command issue aspects. Experienced users look for options such as “hot-keys”, “macros”, etc. Thus, the skill level of users improves as they keep using a software product and they look for commands to suit their skill levels.
- **Error recovery (undo facility).** While issuing commands, even the expert users can commit errors. Therefore, a good user interface should allow a user to undo a mistake committed by him while using the interface. Users are put to inconvenience, if they cannot recover from the errors they commit while using the software.
- **User guidance and on-line help.** Users seek guidance and on-line help when they either forget a command or are unaware of some features of the software. Whenever users need guidance or seek help from the system, they should be provided with the appropriate guidance and help.

9.1.1 User guidance and online help

Users may seek help about the operation of the software any time while using the software. This is provided by the on-line help system. This is different from the guidance and error messages which are flashed automatically without the user asking for them. The guidance messages prompt the user regarding the options he has regarding the next command, and the status of the last command, etc.

On-line Help System. Users expect the on-line help messages to be tailored to the context in which they invoke the “help system”. Therefore, a good on-line help system should keep track of what a user is doing while invoking the help system and provide the output message in a context-dependent way. Also, the help messages should be tailored to the user’s experience level. Further, a good on-line help system should take advantage of any graphics and animation characteristics of the screen and should not just be a copy of the user’s manual. Fig. 9.1 gives a snapshot of a typical on-line help provided by a user interface.

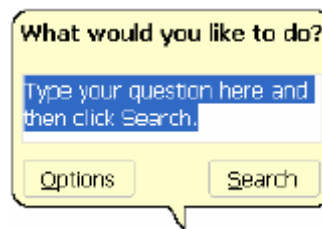


Fig. 9.1. Example of an on-line help interface

Guidance Messages. The guidance messages should be carefully designed to prompt the user about the next actions he might pursue, the current status of the system, the progress made so far in processing his last command, etc. A good guidance system should have different levels of sophistication for different categories of users. For example, a user using a command language interface might need a different type of guidance compared to a user using a menu or iconic interface. Also, users should have an option to turn off detailed messages.

9.1.2 Mode-based interface vs. modeless interface

A mode is a state or collection of states in which only a subset of all user interaction tasks can be performed. In a modeless interface, the same set of commands can be invoked at any time during the running of the software. Thus, a modeless interface has only a single mode and all the commands are available all the time during the operation of the software. On the other hand, in a mode-based interface, different set of commands can be invoked depending on the mode in which the system is, i.e. the mode at any instant is determined by the sequence of commands already issued by the user.

A mode-based interface can be represented using a state transition diagram, where each node of the state transition diagram would represent a mode. Each state of the state transition diagram can be annotated with the commands that are meaningful in that state.

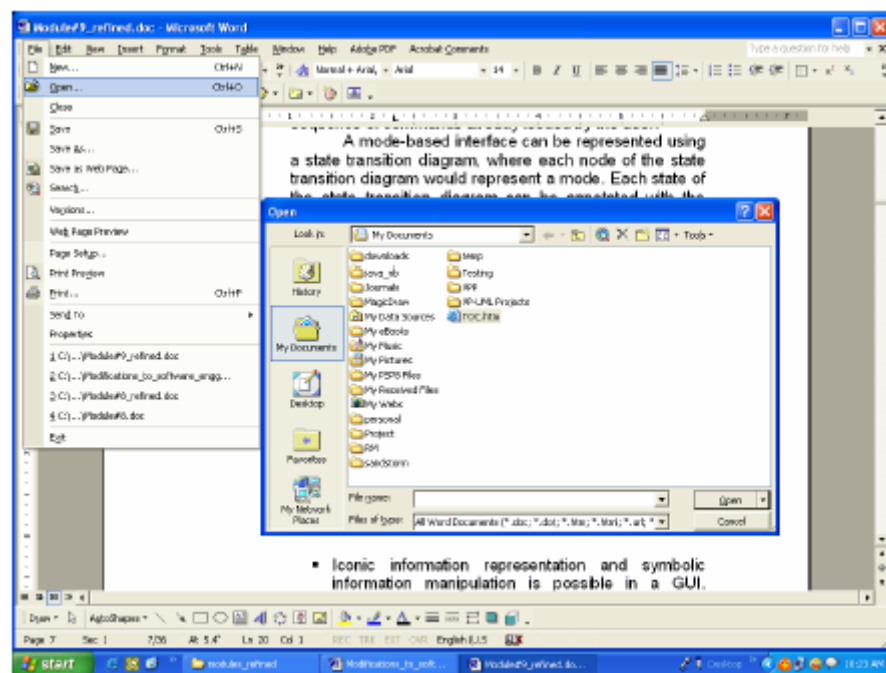


Fig 9.2. An example of mode-based interface

Fig 9.2 shows the interface of a word processing program. The top-level menu provides the user with a gamut of operations like file open, close, save, etc. When the user chooses the open option, another frame is popped up which limits the user to select a name from one of the folders.

9.1.3 Graphical User Interface vs. Text-based User Interface

The following comparisons are based on various characteristics of a GUI with those of a text-based user interface.

- In a GUI multiple windows with different information can simultaneously be displayed on the user screen. This is perhaps one of the biggest advantages of GUI over text-based interfaces since the user has the flexibility to simultaneously interact with several related items at any time and can have access to different system information displayed in different windows.
- Iconic information representation and symbolic information manipulation is possible in a GUI. Symbolic information manipulation such as dragging an icon representing a file to a trash can be deleting is intuitively very appealing and the user can instantly remember it.
- A GUI usually supports command selection using an attractive and user-friendly menu selection system.
- In a GUI, a pointing device such as a mouse or a light pen can be used for issuing commands. The use of a pointing device increases the efficacy issue procedure.

On the flip side, a GUI requires special terminals with graphics capabilities for running and also requires special input devices such a mouse. On the other hand, a text-based user interface can be implemented even on a cheap alphanumeric display terminal. Graphics terminals are usually much more expensive than alphanumeric terminals. However, display terminals with graphics capability with bit-mapped high-resolution displays and significant amount of local processing power have become affordable and over the years have replaced text-based terminals on all desktops. Therefore, the emphasis of this lesson is on GUI design rather than text-based user interface design.

Check Your Progress I

Mark all options which are true.

1. The interface of a software product which is supporting a large number of commands can be developed using

- a. mode-based interface
 - b. modeless interface
 - c. either mode-based interface or modeless interface
 - d. neither mode-based interface nor modeless interface
2. The interface that can be implemented even on cheap alphanumeric terminals is
- a. menu-based interface
 - b. command language-based interface
 - c. iconic interface
 - d. none of them
3. The term “iconic interface” is applicable to
- a. command language-based interface
 - b. menu-based interface
 - c. direct manipulation interface
 - d. none of the above

Mark the following as either True or False.

4. A good user interface must provide feedback to various user actions.
5. A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users.
6. In the context of user interface, “Guidance” and “On-line Help” are used for the same purpose.

9.2 TYPES OF USER INTERFACES

User interfaces can be classified into the following three categories:

- Command language-based interfaces
- Menu-based interfaces
- Direct manipulation interfaces

9.2.1 Command Language-based Interface

A command language-based interface – as the name itself suggests, is based on designing a command language which the user can use to issue the commands. The user is expected to frame the appropriate commands in the language and type

them in appropriately whenever required. A simple command language-based interface might simply assign unique names to the different commands. However, a more sophisticated command language-based interface may allow users to compose complex commands by using a set of primitive commands. Such a facility to compose commands dramatically reduces the number of command names one would have to remember. Thus, a command language-based interface can be made concise requiring minimal typing by the user. Command language-based interfaces allow fast interaction with the computer and simplify the input of complex commands.

9.2.2 Direct Manipulation Interfaces

Direct manipulation interfaces present the interface to the user in the form of visual models (i.e. icons or objects). For this reason, direct manipulation interfaces are sometimes called as iconic interface. In this type of interface, the user issues commands by performing actions on the visual representations of the objects, e.g. pull an icon representing a file into an icon representing a trash box, for deleting the file. Important advantages of iconic interfaces include the fact that the icons can be recognized by the users very easily, and that icons are language-independent. However, direct manipulation interfaces can be considered slow for experienced users. Also, it is difficult to give complex commands using a direct manipulation interface. For example, if one has to drag an icon representing the file to a trash box icon for deleting a file, then in order to delete all the files in the directory one has to perform this operation individually for all files – which could be very easily done by issuing a command like `delete *.*`.

9.2.3 Menu-based Interface

An important advantage of a menu-based interface over a command language-based interface is that a menu-based interface does not require the users to remember the exact syntax of the commands. A menu-based interface is based on recognition of the command names, rather than recollection. Further, in a menu-based interface the typing effort is minimal as most interactions are carried out

through menu selections using a pointing device. This factor is an important consideration for the occasional user who cannot type fast.

However, experienced users find a menu-based user interface to be slower than a command language-based interface because an experienced user can type fast and can get speed advantage by composing different primitive commands to express complex commands. Composing commands in a menu-based interface is not possible. This is because of the fact that actions involving logical connectives (and, or, etc.) are awkward to specify in a menu-based system. Also, if the number of choices is large, it is difficult to select from the menu. In fact, a major challenge in the design of a menu-based interface is to structure large number of menu choices into manageable forms.

When the menu choices are large, they can be structured as the following way:

Scrolling menu

When a full choice list cannot be displayed within the menu area, scrolling of the menu items is required. This would enable the user to view and select the menu items that cannot be accommodated on the screen. However, in a scrolling menu all the commands should be highly correlated, so that the user can easily locate a command that he needs. This is important since the user cannot see all the commands at any one time. An example situation where a scrolling menu is frequently used is font size selection in a document processor (as shown in fig. 9.3). Here, the user knows that the command list contains only the font sizes that are arranged in some order and he can scroll up and down to find the size he is looking for. However, if the commands do not have any definite ordering relation, then the user would have to in the worst case, scroll through all the commands to find the exact command he is looking for, making this organization inefficient.

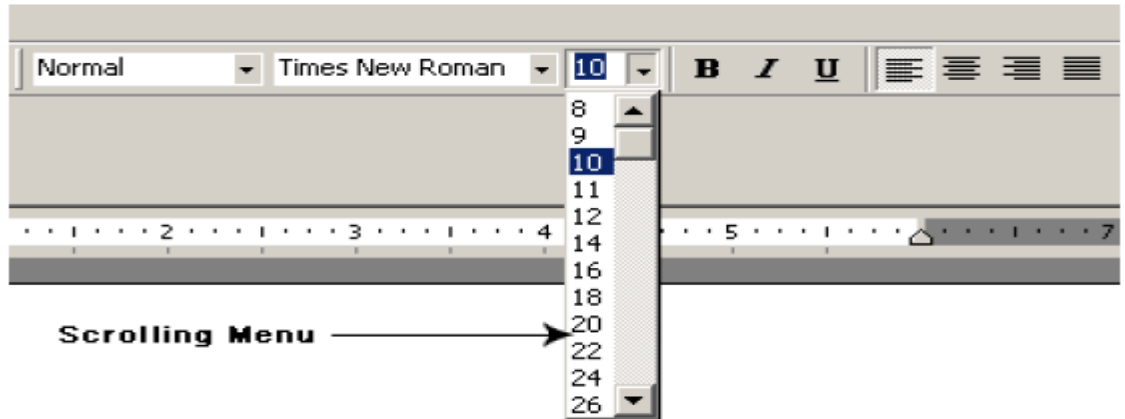


Fig. 9.3: Font size selection using scrolling menu

Walking menu

Walking menu is very commonly used to structure a large collection of menu items. In this technique, when a menu item is selected, it causes further menu items to be displayed adjacent to it in a sub-menu. An example of a walking menu is shown in fig. 9.4. A walking menu can successfully be used to structure commands only if there are tens rather than hundreds of choices since each adjacently displayed menu does take up screen space and the total screen area is after limited.

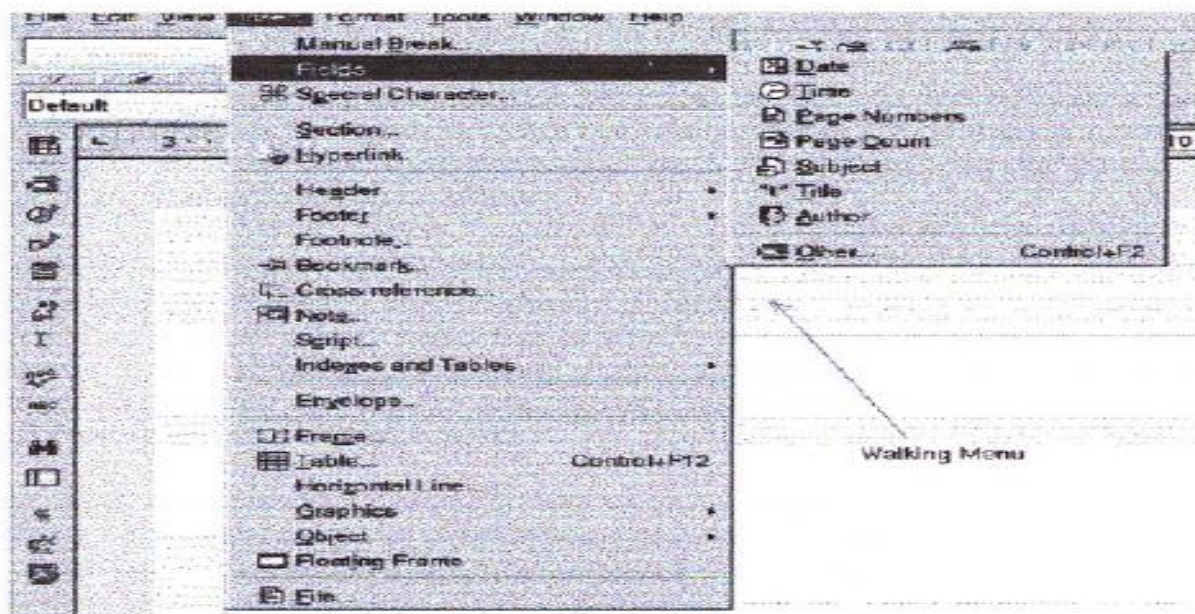


Fig. 9.4: Example of walking menu

Hierarchical menu

In this technique, the menu items are organized in a hierarchy or tree structure. Selecting a menu item causes the current menu display to be replaced by an appropriate sub-menu. Thus in this case, one can consider the menu and its various sub-menus to form a hierarchical tree-like structure. Walking menu can be considered to be a form of hierarchical menu which is practicable when the tree is shallow. Hierarchical menu can be used to manage large number of choices, but the users are likely to face navigational problems because they might lose track of where they are in the menu tree. This probably is the main reason why this type of interface is very rarely used.

Characteristics of command language-based interface

Characteristics of command language-based interface have been discussed earlier.

Disadvantages of command language-based interface

Command language-based interfaces suffer from several drawbacks. Usually, command language-based interfaces are difficult to learn and require the user to memorize the set of primitive commands. Also, most users make errors while formulating commands in the command language and also while typing them in. Further, in a command language-based interface, all interactions with the system is through a key-board and can not take advantage of effective interaction devices such as a mouse. Obviously, for casual and inexperienced users, command language-based interfaces are not suitable.

Issues in designing a command language-based interface

Two overbearing command design issues are to reduce the number of primitive commands that a user has to remember and to minimize the total typing required while issuing commands. These can be elaborated as follows:

- The designer has to decide what mnemonics are to be used for the different commands. The designer should try to develop meaningful mnemonics and yet be concise to minimize the amount of typing required. For example, the shortest mnemonic should be assigned to the most frequently used commands.
- The designer has to decide whether the users will be allowed to redefine the command names to suit their own preferences. Letting a user define his own mnemonics for various commands is a useful feature, but it increases the complexity of user interface development.

- The designer has to decide whether it should be possible to compose primitive commands to form more complex commands. A sophisticated command composition facility would require the syntax and semantics of the various command composition options to be clearly and unambiguously specified. The ability to combine commands is a powerful facility in the hands of experienced users, but quite unnecessary for inexperienced users.

9.2.4 Types of menus and their features

Three main types of menus are scrolling menu, walking menu, and hierarchical menu. The features of scrolling menu, walking menu, and hierarchical menu have been discussed earlier.

Iconic interface

Direct manipulation interfaces present the interface to the user in the form of visual models (i.e. icons or objects). For this reason, direct manipulation interfaces are sometimes called iconic interfaces. In this type of interface, the user issues commands by performing actions on the visual representations of the objects, e.g. pull an icon representing a file into an icon representing a trash box, for deleting the file.



Fig 9.5. Example of an iconic interface

Fig 9.5 shows an iconic interface. Here, the user is presented with a set of icons at the top of the frame for performing various activities. On clicking on any of the icons, either the user is prompted with a sub menu or the desired activity is performed.

Component-based GUI development

A development style based on widgets (window objects) is called component-based (or widget-based) GUI development style. There are several important advantages of using a widget-based design style. One of the most important reasons to use widgets as building blocks is because they help users learn an interface fast. In this style of development, the user interfaces for different applications are built from the same basic components. Therefore, the user can extend his knowledge of the behavior of the standard components from one application to the other. Also, the component-based user interface development style reduces the application programmer's work significantly as he is more of a user interface component integrator than a programmer in the traditional sense.

Need for component-based GUI development

The current style of user interface development is component-based. It recognizes that every user interface can easily be built from a handful of predefined components such as menus, dialog boxes, forms, etc. Besides the standard components, and the facilities to create good interfaces from them, one of the basic support available to the user interface developers is the window system. The window system lets the application programmer create and manipulate windows without having to write the basic windowing functions.

9.3 COMPONENT-BASED GUI DEVELOPMENT

9.3.1 Window

A window is a rectangular area on the screen. A window can be considered to be a virtual screen, in the sense that it provides an interface to the user for carrying out independent activities, e.g. one window can be used for editing a program and another for drawing pictures, etc.

A window can be divided into two parts: client part, and non-client part. The client area makes up the whole of the window, except for the borders and scroll bars. The client area is the area available to a client application for display. The non-client part of the window determines the look and feel of the window. The look and feel defines a basic behavior for all windows, such as creating, moving, resizing, and iconifying the windows. A basic window with its different parts is shown in fig. 9.6.

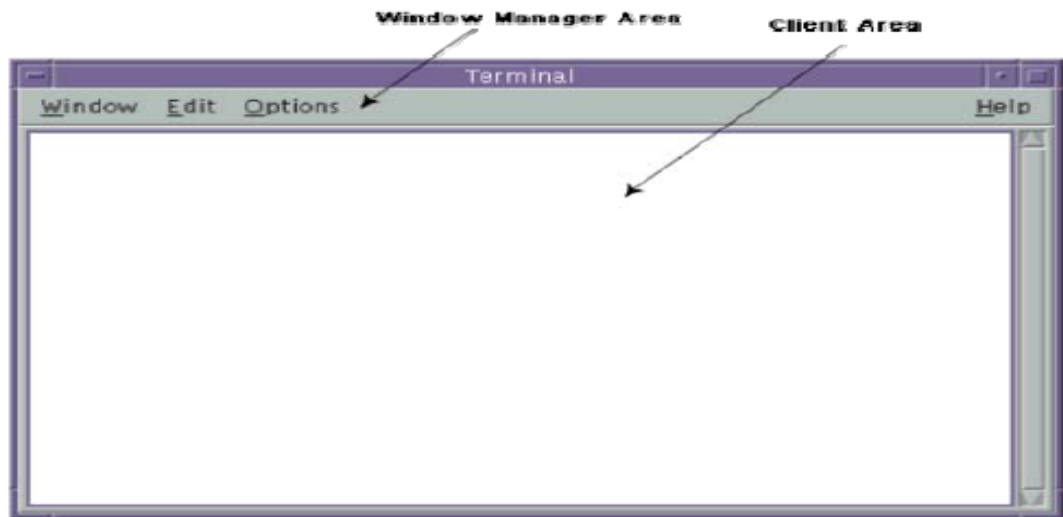


Fig. 9.6: Window with client and user areas marked

9.3.2 Window Management System (WMS)

A graphical user interface typically consists of a large number of windows. Therefore, it is necessary to have some systematic way to manage these windows. Most graphical user interface development environments do this through a window management system (WMS). A window management system is primarily a resource manager. It keeps track of the screen area resource and allocates it to the different windows that seek to use the screen. From a broader perspective, a WMS can be considered as a user interface management system (UIMS) – which not only does resource management, but also provides the basic behavior to the windows and provides several utility routines to the application programmer for user interface development. A WMS simplifies the task of a GUI designer to a great extent by providing the basic behavior to the various windows such as move, resize, iconify, etc. as soon as they are created and by providing the basic routines

to manipulate the windows from the application such as creating, destroying, changing different attributes of the windows, and drawing text, lines, etc.

A WMS consists of two parts (as shown in fig. 9.7):

- a window manager, and
- a window system.

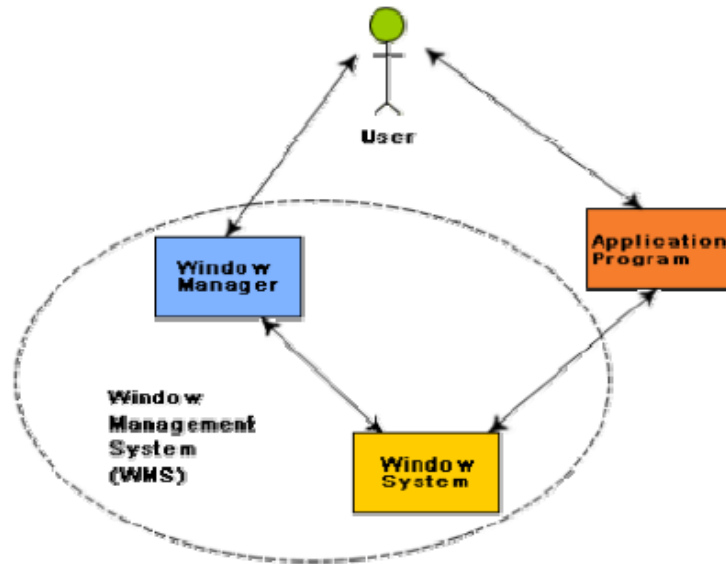


Fig. 9.7: Window Management System

9.3.3 Window Manager and Window System

Window manager is the component of WMS with which the end user interacts to do various window-related operations such as window repositioning, window resizing, iconification, etc. The window manager is built on the top of the window system in the sense that it makes use of various services provided by the window system. The window manager and not the window system determine how the windows look and behave. In fact, several kinds of window managers can be developed based on the same window system. The window manager can be considered as a special kind of client that makes use of the services (function calls) supported by the window system. The application programmer can also directly invoke the services of the window system to develop the user interface. The relationship between the window manager, window system, and the application program is shown in [fig. 9.7](#). This figure shows that the end-user can

either interact with the application itself or with the window manager (resize, move, etc.) and both the application and the window manager invoke services of the window manager.

9.3.3.1 Window Manager

The window manager is responsible for managing and maintaining the non-client area of a window. Window manager manages the real-estate policy, provides look and feel of each individual window.

Types of widgets (window objects)

Different interface programming packages support different widget sets. However, a surprising number of them contain similar kinds of widgets, so that one can think of a generic widget set which is applicable to most interfaces. The following widgets are representatives of this generic class.

Label widget. This is probably one of the simplest widgets. A label widget does nothing except to display a label, i.e. it does not have any other interaction capabilities and is not sensitive to mouse clicks. A label widget is often used as a part of other widgets.

Container widget. These widgets do not stand by themselves, but exist merely to contain other widgets. Other widgets are created as children of the container widget. When the container widget is moved or resized, its children widget also get moved or resized. A container widget has no callback routines associated with it.

Pop-up menu. These are transient and task specific. A pop-up menu appears upon pressing the mouse button, irrespective of the mouse position.

Pull-down menu. These are more permanent and general. You have to move the cursor to a specific location and pull down this type of menu.

Dialog boxes. We often need to select multiple elements from a selection list. A dialog box remains visible until explicitly dismissed by the user. A dialog box can include areas for entering text as well as values. If an apply command is supported in a dialog box, the newly entered values can be tried without dismissing the box. Through most dialog boxes ask you to enter some information, there are some dialog boxes which are merely informative, alerting you to a problem with your system or an error you have made.

Generally, these boxes ask you to read the information presented and then click *OK* to dismiss the box.

Push button. A push button contains key words or pictures that describe the action that is triggered when you activate the button. Usually, the action related to a push button occurs immediately when you click a push button unless it contains an ellipsis (...). A push button with an ellipsis generally indicates that another dialog box will appear.

Radio buttons. A set of radio buttons is used when only one option has to be selected out of many options. A radio button is a hollow circle followed by text describing the option it stands for. When a radio button is selected, it appears filled and the previously selected radio button from the group is unselected. Only one radio button from a group can be selected at any time. This operation is similar to that of the band selection buttons that were available in old radios.

Combo boxes. A combo box looks like a button until the user interacts with it. When the user presses or clicks it, the combo box displays a menu of items to choose from. Normally a combo box is used to display either one-of-many choices when space is limited, the number of choices is large, or when the menu items are computed at run-time.

9.3.4 X-Window

The X-window functions are low-level functions written in C language which can be called from application programs. But only the very serious application designer would program directly using the X-windows library routines. Built on top of X-windows are higher-level functions collectively called Xtoolkit. Xtoolkit consists of a set of basic widgets and a set of routines to manipulate these widgets. One of the most widely used widget sets is X/Motif. Digital Equipment Corporation (DEC) used the basic X-window functions to develop its own look and feel for interface designs called DECWindows.

9.3.4.1 Popularity of X-Window

One of the important reasons behind the extreme popularity of the X-window system is probably due to the fact that it allows development of portable GUIs.

Applications developed using X-window system are device-independent. Also, applications developed using the X-window system become network independent in the sense that the interface would work just as well on a terminal connected anywhere on the same network as the computer running the application is. Network-independent GUI operation has been schematically represented in the fig. 9.8. Here, “A” is the computer application in which the application is running. “B” can be any computer on the network from where interaction with the application can be made. Network-independent GUI was pioneered by the X-window system in the mid-eighties at MIT (Massachusetts Institute of Technology) with support from DEC (Digital Equipment Corporation). Now-a-days many user interface development systems support network-independent GUI development, e.g. the AWT and Swing components of Java.

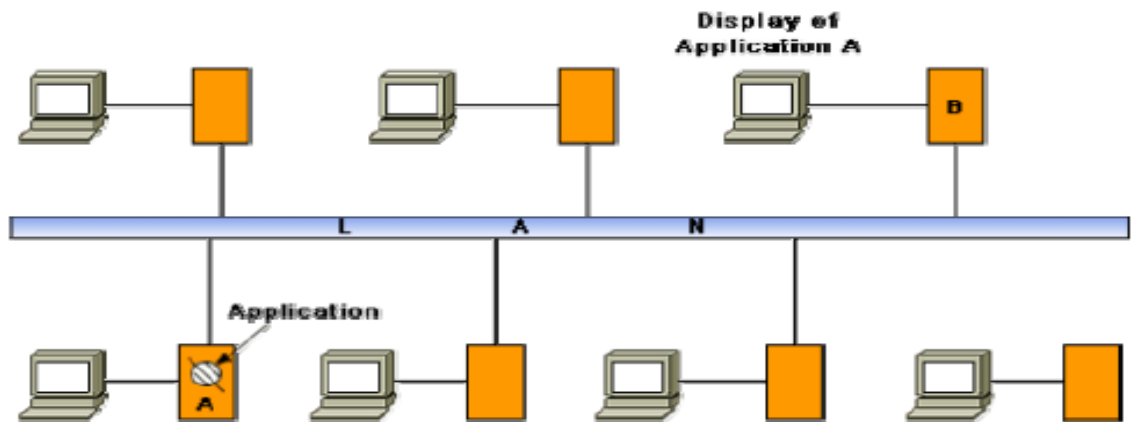


Fig. 9.8: Network-independent GUI

9.3.4.2 Architecture of an X-System

The X-architecture is pictorially depicted in fig. 9.9. The different terms used in this diagram are explained below.

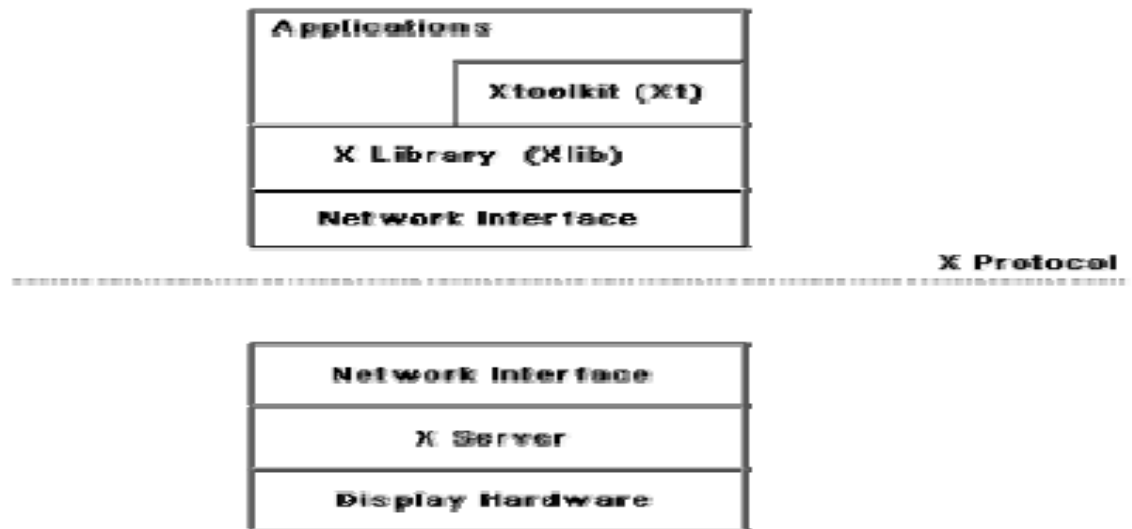


Fig. 9.9: Architecture of the X-System

X-server. The X server runs on the hardware to which the display and keyboard attached. The X server performs low-level graphics, manages window, and user input functions. The X server controls accesses to a bit-mapped graphics display resource and manages it.

X-protocol. The X protocol defines the format of the requests between client applications and display servers over the network. The X protocol is designed to be independent of hardware, operating systems, underlying network protocol, and the programming language used.

X-library (Xlib). The Xlib provides a set of about 300 utility routines for applications to call. These routines convert procedure calls into requests that are transmitted to the server. Xlib provides low level primitives for developing an user interface, such as displaying a window, drawing characteristics and graphics on the window, waiting for specific events, etc.

Xtoolkit (Xt). The Xtoolkit consists of two parts: the intrinsics and the widgets. We have already seen that widgets are predefined user interface components such as scroll bars, push buttons, etc. for designing GUIs. Intrinsics are a set of about a dozen library routines that allow a programmer to combine a set of widgets into a user interface. In order to develop a user interface, the designer has to put together the set of components (widgets) he needs, and then he needs to define the characteristics (called resources) and behavior of these widgets by using the intrinsic routines to complete the development of the interface. Therefore,

developing an interface using Xtoolkit is much easier than developing the same interface using only X library.

9.3.5 Visual Programming

Visual programming is the drag and drop style of program development. In this style of user interface development, a number of visual objects (icons) representing the GUI components are provided by the programming environment. The application programmer can easily develop the user interface by dragging the required component types (e.g. menu, forms, etc.) from the displayed icons and placing them wherever required. Thus, visual programming can be considered as program development through manipulation of several visual objects. Reuse of program components in the form of visual objects is an important aspect of this style of programming. Though popular for user interface development, this style of programming can be used for other applications such as Computer-Aided Design application (e.g. factory design), simulation, etc. User interface development using a visual programming language greatly reduces the effort required to develop the interface.

Examples of popular visual programming languages are Visual Basic, Visual C++, etc. Visual C++ provides tools for building programs with window-based user interfaces for Microsoft Windows environments. In Visual C++, menu bars, icons, and dialog boxes, etc. can be designed easily before adding them to program. These objects are called as resources. Shape, location, type, and size of the dialog boxes can be designed before writing any C++ code for the application.

Difference between user-centered design and design by users

- User-centered design is the theme of almost all modern user interface design techniques. However, user-centered design does not mean design by users. One should not get the users to design the interface, nor should one assume that the user's opinion of which design alternative is superior is always right.
- Users have good knowledge of the tasks they have to perform, they also know whether they find an interface easy to learn and use but they have less understanding and experience in GUI design than the GUI developers.

Implications of human cognition capabilities on user interface design

An area of human-computer interaction where extensive research has been conducted is how human cognitive capabilities and limitations influence the way an interface should be designed. The following are some of the prominent issues extensively discussed in the literature.

- **Limited memory.** Humans can remember at most seven unrelated items of information for short periods of time. Therefore, the GUI designer should not require the user to remember too many items of information at a time. It is the GUI designer's responsibility to anticipate what information the user will need at what point of each task and to ensure that the relevant information is displayed for the user to see. Showing the user some information at some point, and then asking him to recollect that information in a different screen where they no longer see the information places a memory burden on the user and should be avoided wherever possible.
- **Frequent task closure.** Doing a task (except for very trivial tasks) requires doing several subtasks. When the system gives a clear feedback to the user that a task has been successfully completed, the user gets a sense of achievement and relief. The user can clear out information regarding the completed task from memory. This is known as task closure. When the overall task is fairly big and complex, it should be divided into subtasks, each of which has a clear subgoal which can be a closure point.
- **Recognition rather than recall.** Information recall incurs a larger memory burden on the users and is to be avoided as far as possible. On the other hand, recognition of information from the alternatives shown to him is more acceptable.
- **Procedural versus object-oriented.** Procedural designs focus on tasks, prompting the user in each step of the task, giving them few options for anything else. This approach is the best applied in situations where the tasks are narrow and well-defined or where the users are inexperienced, such as an ATM. An object-oriented interface on the other hand focuses on objects. This allows the users a wide range of options.

9.3.6 GUI design methodology

GUI design methodology consists of the following important steps:

- Examine the use case model of the software. Interview, discuss, and review the GUI issues with the end-users.
- Task and object modeling
- Metaphor selection
- Interaction design and rough layout
- Detailed presentation and graphics design
- GUI construction
- Usability evaluation

The starting point for GUI design is the use case model. This captures the important tasks the users need to perform using the software. As far as possible, a user interface should be developed using one or more metaphors. Metaphors help in interface development at lower effort and reduced costs for training the users. Over time, people have developed efficient methods of dealing with some commonly occurring situations. These solutions are the themes of the metaphors. Metaphors can also be based on physical objects such as a visitor's book, a catalog, a pen, a brush, a scissor, etc. A solution based on metaphors is easily understood by the users, reducing learning time and training costs. Some commonly used metaphors are the following:

- White board
- Shopping cart
- Desktop
- Editor's work bench
- White page
- Yellow page
- Office cabinet
- Post box
- Bulletin board
- Visitor's book

Task and Object Modeling

A task is a human activity intended to achieve some goals. Example of task goals can be:

- reserve an airline seat
- buy an item
- transfer money from one account to another
- book a cargo for transmission to an address

A task model is an abstract model of the structure of a task. A task model should show the structure of the subtasks that the user needs to perform to achieve the overall task goal. Each task can be modeled as a hierarchy of subtasks. A task model can be drawn using a graphical notation similar to the activity network model. Tasks can be drawn as boxes with lines showing how a task is broken down into subtasks. An underlined task box would mean that no further decomposition of the task is required. An example decomposition of a task into subtasks is shown in fig. 9.10.

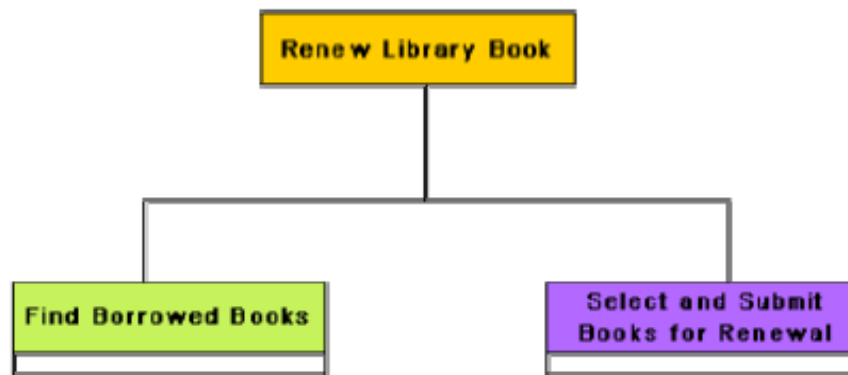


Fig. 9.10: Decomposition of a task into subtasks

Selecting a metaphor

The first place one should look for while trying to identify the candidate metaphors is the set of parallels to objects, tasks, and terminologies of the use cases. If no obvious metaphors can be found, then the designer can fall back on the metaphors of the physical world of concrete objects. The appropriateness of each candidate metaphor should be tested by restating the objects and tasks of the user interface model in terms of the metaphor. Another criterion that can be used to judge metaphors is that the metaphor should be as simple as possible, the

operations using the metaphor should be clear and coherent and it should fit with the users' 'common sense' knowledge. For example, it would indeed be very awkward and a nuisance for the users if the scissor metaphor is used to glue different items.

Example: We need to develop the interface for the automation shop, where the users can examine the contents of the shop through a web interface and can order them.

Several metaphors are possible for different parts of this problem.

- Different items can be picked up from **racks** and examined. The user can request for the **catalog** associated with the items by clicking on the item.
- Related items can be picked from the **drawers** of an item cabinet.
- The items can be organized in the form of a book, similar to the way information about electronic components are organized in a semiconductor hand book.

Once the users make up their mind about an item they wish to buy, they can put them into a **shopping cart**.

User interface inspection

Nielson [Niel94] studied common usability problems and built a check list of points which can be easily checked for an interface. The following check list is based on the work of Nielson [Niel94].

Visibility of the system status. The system should as far as possible keep the user informed about the status of the system and what is going on.

Match between the system and the real world. The system should speak the user's language words, phrases, and concepts familiar to that used by the user, rather than using system-oriented terms.

Undoing mistakes. The user should feel that he is in control rather than feeling helpless or to be at the control of the system. An important step toward this is that the users should be able to undo and redo operations.

Consistency. The user should not have to wonder whether different words, concepts, and operations mean the same thing in different situations.

Recognition rather than recall. The user should not have to recall information which was presented in another screen. All data and instructions should be visible on the screen for selection by the user.

Support for multiple skill levels. Provision of accelerations for experienced users allows them to efficiently carry out the actions they frequently require to perform.

Aesthetic and minimalist design. Dialogs should not contain information which are irrelevant and are rarely needed. Every extra unit of information in a dialog competes with the relevant units and diminishes their visibility.

Help and error messages. These should be expressed in plain language (no codes), precisely indicating the problem, and constructively suggesting a solution.

Error prevention. Error possibilities should be minimized. A key principle in this regard is to prevent the user from entering wrong values. In situations where a choice has to be made from among a discrete set of values, the control should present only the valid values using a drop-down list, a set of option buttons or a similar multichoice control. When a specific format is required for attribute data, the entered data should be validated when the user attempts to submit the data.

Check Your Progress II

1. A command composition facility can be made available in case of
 - a. menu-based interface
 - b. command language-based interface
 - c. direct manipulation interface
 - d. none of the above
2. A development style based on widgets is called
 - a. command language-based GUI development style
 - b. component-based GUI development style
 - c. menu-based GUI development style
 - d. direct manipulation based GUI development style
3. In the case of X Window architecture, the format of the requests between client applications and display servers over the network is defined by
 - a. X-server
 - b. X-library
 - c. X-protocol

- d. Xtoolkit
- 4. The low level primitives for developing a user interface, such as displaying a window, drawing characteristics and graphics on the window etc. in case of X Window system are provided by
 - a. X-server
 - b. X-library
 - c. X-protocol
 - d. Xtoolkit

Mark the following as either True or False.

- 5. Novice users normally prefer command language interfaces over both menu-based and iconic interfaces.
- 6. Intrinsic of Xtoolkit of a X Window system are a set of about dozen library routines that allow a programmer/user interface designer to combine a set of widgets into a user interface.
- 7. Visual programming style is restricted to user interface development only.
- 8. Visual programming can be considered as program development through manipulation of several visual objects.
- 9. For modern user interfaces, LOC is an accurate measure of the size of the interface.
- 10. When a window is a modal dialog, no other windows in the application are accessible until the current window is closed.
- 11. Graphical User Interfaces should let the user recall commands rather than have him recognize commands from a repertoire of displayed commands.
- 12. In case of good user interface design, the GUI designer should try to reduce the number of screens by cramming as much information on a screen as possible.

9.4 SUMMARY

- 1. It is very important to identify the characteristics desired of a good user interface.
- 2. A good user interface should be easy to learn. Speed of learning is hampered by complex syntax and semantics of the command issue procedures.

3. Speed of learning an interface is greatly facilitated if these are based on some day-to-day real-life examples or some physical objects with which the users are familiar.
4. The abstractions of real-life objects or concepts used in user interface design are called metaphors.
5. Once a user learns about a command, he should be able to use the similar commands in different circumstances for carrying out similar actions.
6. Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other applications with which the user is already familiar.
7. User interfaces can be classified into the following three categories: Command language based interfaces, Menu-based interfaces and Direct manipulation interfaces.
8. A development style based on widgets (window objects) is called component-based (or widget-based) GUI development style.
9. Visual programming is the drag and drop style of program development. In this style of user interface development, a number of visual objects (icons) representing the GUI components are provided by the programming environment.

9.5 MODEL QUESTIONS

1. List five desirable characteristics that a good user interface should possess.
2. What is the difference between user guidance and on-line help system in the user interface of a software system?
3. Discuss the different ways in which on-line help can be provided to a user while he is executing the software.
4. What is the difference between a mode-based interface and a modeless interface?
5. Compare the relative advantages of textual and graphical user interfaces.
6. Compare the relative advantages of command language, menu-based, and direct manipulation interfaces.
7. List the important advantages and disadvantages of a command language interface.

8. List the important advantages and disadvantages of a menu-based interface.
9. Compare the relative advantages of scrolling menu, hierarchical menu, and walking menu as techniques for organizing user commands.
10. What do you understand by an iconic interface? Explain how you can issue commands using an iconic interface.
11. List the important advantages and disadvantages of a direct manipulation interface.
12. Suppose you have been asked to design the user interface of a large software product. Would you choose a menu-based, a direct manipulation, a command language-based, or a mixture of all these types of interfaces to develop the interface for your product? Justify your choice.
13. Explain the reason of popularity of component-based GUI development.
14. What is Window Management System (WMS)? Represent the main components of a WMS in a schematic diagram and explain their roles.
15. What are the advantages of using a Window Management System (WMS) for a GUI design? Name some commercially available Window Management Systems.
16. Explain the responsibilities of a window manager in the context of Window Management System (WMS).
17. Discuss the architecture of the X window system.
18. What are the important advantages of using the X window system for developing graphical user interfaces?
19. What do you understand by visual programming?
20. Distinguish between a user-centric interface design and interface design by users.
21. How does the human cognition capabilities and limitations influence human-computer user interface designing?
22. What do you understand by a metaphor in a user interface design? Is a metaphor-based user interface design advantageous? Justify it.
23. List a few metaphors which can be used for user interface design.
24. What is meant by a task model? Explain it with examples.
25. Do prepare a check list for user interface inspection.

UNIT X: CODING AND TESTING

10.0 LEARNING OBJECTIVES

At the end of this lesson the student would be able to:

- Identify the necessity of coding standards.
- Differentiate between coding standards and coding guidelines.
- State what code review is.
- Explain what clean room testing is.
- Explain the necessity of properly documenting software.
- Differentiate between internal documentation and external documentation.
- Explain what is testing.
- Explain the aim of testing.
- Differentiate between verification and validation.
- Explain why random selection of test cases is not effective.
- Differentiate between functional testing and structural testing.
- Differentiate between testing in the large and testing in the small.
- Explain what unit testing.
- Explain what black box testing is.
- Identify equivalence classes for any given problem.
- Explain what is meant by boundary value analysis.
- Design test cases corresponding to equivalence class testing and boundary value analysis for any given problem.
- In the context of white box testing strategy, differentiate between stronger testing and complementary testing.
- Design statement coverage test cases for a code segment.
- Design branch coverage test cases for a code segment.
- Design condition coverage test cases for a code segment.
- Design path coverage test cases for a code segment.
- Draw control flow graph for any program.
- Identify the linear independent paths.
- Compute cyclomatic complexity from any control flow graph.

- Explain data flow-based testing.
- Explain mutation testing.
- Explain why debugging is needed.
- Explain three approaches of debugging.
- Explain three guidelines for effective debugging.
- Explain what is meant by a program analysis tool.
- Explain the functions of a static program analysis tool.
- Explain the functions of a dynamic program analysis tool.
- Explain the type of failures detected by integration testing.
- Identify four types of integration test approaches and explain them.
- Differentiate between phased and incremental testing in the context of integration testing.
- What are three types of system testing? Differentiate among them.
- Identify nine types of performance tests that can be performed to check whether the system meets the non-functional requirements identified in the SRS document.
- Explain what is meant by error seeding.
- Explain what functions are performed by regression testing.

10.1 CODE REVIEW

10.1.1 Coding

Good software development organizations normally require their programmers to adhere to some well-defined and standard style of coding called coding standards. Most software development organizations formulate their own coding standards that suit them most, and require their engineers to follow these standards rigorously. The purpose of requiring all engineers of an organization to adhere to a standard style of coding is the following:

- A coding standard gives a uniform appearance to the codes written by different engineers.
- It enhances code understanding.

- It encourages good programming practices.

A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, error return conventions, etc.

10.1.1.1 Coding standards and guidelines

Good software development organizations usually develop their own coding standards and guidelines depending on what best suits their organization and the type of products they develop.

The following are some representative coding standards.

Rules for limiting the use of global: These rules list what types of data can be declared global and what cannot.

Contents of the headers preceding codes for different modules: The information contained in the headers of different modules should be standard for an organization. The exact format in which the header information is organized in the header can also be specified. The following are some standard header data:

- Name of the module.
- Date on which the module was created.
- Author's name.
- Modification history.
- Synopsis of the module.
- Different functions supported, along with their input/output parameters.
- Global variables accessed/modified by the module.

Naming conventions for global variables, local variables, and constant identifiers: A possible naming convention can be that global variable names always start with a capital letter, local variable names are made of small letters, and constant names are always capital letters.

Error return conventions and exception handling mechanisms: The way error conditions are reported by different functions in a program are handled should be standard within an organization. For example, different functions while encountering an error condition should either return a 0 or 1 consistently.

The following are some representative coding guidelines recommended by many software development organizations.

Do not use a coding style that is too clever or too difficult to understand: Code should be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code. Clever coding can obscure meaning of the code and hamper understanding. It also makes maintenance difficult.

Avoid obscure side effects: The side effects of a function call include modification of parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code. For example, if a global variable is changed obscurely in a called module or some file I/O is performed which is difficult to infer from the function's name and header information, it becomes difficult for anybody trying to understand the code.

Do not use an identifier for multiple purposes: Programmers often use the same identifier to denote several temporary entities. For example, some programmers use a temporary loop variable for computing and a storing the final result. The rationale that is usually given by these programmers for such multiple uses of variables is memory efficiency, e.g. three variables use up three memory locations, whereas the same variable used in three different ways uses just one memory location. However, there are several things wrong with this approach and hence should be avoided. Some of the problems caused by use of variables for multiple purposes as follows:

- Each variable should be given a descriptive name indicating its purpose. This is not possible if an identifier is used for multiple purposes. Use of a variable for multiple purposes can lead to confusion and make it difficult for somebody trying to read and understand the code.
- Use of variables for multiple purposes usually makes future enhancements more difficult.

The code should be well-documented: As a rule of thumb, there must be at least one comment line on the average for every three-source line.

The length of any function should not exceed 10 source lines: A function that is very lengthy is usually very difficult to understand as it probably carries out many different functions. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.

Do not use goto statements: Use of goto statements makes a program unstructured and makes it very difficult to understand.

10.1.2 Code review

Code review for a model is carried out after the module is successfully compiled and the all the syntax errors have been eliminated. Code reviews are extremely cost-effective strategies for reduction in coding errors and to produce high quality code. Normally, two types of reviews are carried out on the code of a module. These two types code review techniques are code inspection and code walk through.

10.1.2.1 Code Walk Throughs

Code walk through is an informal code analysis technique. In this technique, after a module has been coded, successfully compiled and all syntax errors eliminated. A few members of the development team are given the code few days before the walk-through meeting to read and understand code. Each member selects some test cases and simulates execution of the code by hand (i.e. trace execution through each statement and function execution). The main objectives of the walk through are to discover the algorithmic and logical errors in the code. The members note down their findings to discuss these in a walk-through meeting where the coder of the module is present.

Even though a code walk through is an informal analysis technique, several guidelines have evolved over the years for making this naïve but useful analysis technique more effective. Of course, these guidelines are based on personal experience, common sense, and several subjective factors. Therefore, these guidelines should be considered as examples rather than accepted as rules to be applied dogmatically. Some of these guidelines are the following.

- The team performing code walk through should not be either too big or too small. Ideally, it should consist of between three to seven members.
- Discussion should focus on discovery of errors and not on how to fix the discovered errors.

- In order to foster cooperation and to avoid the feeling among engineers that they are being evaluated in the code walk through meeting, managers should not attend the walk-through meetings.

10.1.2.2 Code Inspection

In contrast to code walk through, the aim of code inspection is to discover some common types of errors caused due to oversight and improper programming. In other words, during code inspection the code is examined for the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code walk throughs. For instance, consider the classical error of writing a procedure that modifies a formal parameter while the calling routine calls that procedure with a constant actual parameter. It is more likely that such an error will be discovered by looking for these kinds of mistakes in the code, rather than by simply hand simulating execution of the procedure. In addition to the commonly made errors, adherence to coding standards is also checked during code inspection. Good software development companies collect statistics regarding different types of errors commonly committed by their engineers and identify the type of errors most frequently committed. Such a list of commonly committed errors can be used during code inspection to look out for possible errors.

Following is a list of some classical programming errors which can be checked during code inspection:

- Use of uninitialized variables.
- Jumps into loops.
- Nonterminating loops.
- Incompatible assignments.
- Array indices out of bounds.
- Improper storage allocation and deallocation.
- Mismatches between actual and formal parameter in procedure calls.
- Use of incorrect logical operators or incorrect precedence among operators.
- Improper modification of loop variables.
- Comparison of equality of floating-point variables, etc.

10.1.3 Clean room testing

Clean room testing was pioneered by IBM. This type of testing relies heavily on walk throughs, inspection, and formal verification. The programmers are not allowed to test any of their code by executing the code other than doing some syntax testing using a compiler. The software development philosophy is based on avoiding software defects by using a rigorous inspection process. The objective of this software is zero-defect software.

The name 'clean room' was derived from the analogy with semi-conductor fabrication units. In these units (clean rooms), defects are avoided by manufacturing in ultra-clean atmosphere. In this kind of development, inspections to check the consistency of the components with their specifications has replaced unit-testing.

This technique reportedly produces documentation and code that is more reliable and maintainable than other development methods relying heavily on code execution-based testing.

The clean room approach to software development is based on five characteristics:

- **Formal specification:** The software to be developed is formally specified. A state-transition model which shows system responses to stimuli is used to express the specification.
- **Incremental development:** The software is partitioned into increments which are developed and validated separately using the clean room process. These increments are specified, with customer input, at an early stage in the process.
- **Structured programming:** Only a limited number of control and data abstraction constructs are used. The program development process is process of stepwise refinement of the specification.
- **Static verification:** The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components.
- **Statistical testing of the system:** The integrated software increment is tested statistically to determine its reliability. These statistical tests are

based on the operational profile which is developed in parallel with the system specification.

The main problem with this approach is that testing effort is increased as walk throughs, inspection, and verification are time-consuming.

10.1.4 Software documentation

When various kinds of software products are developed then not only the executable files and the source code are developed but also various kinds of documents such as users' manual, software requirements specification (SRS) documents, design documents, test documents, installation manual, etc are also developed as part of any software engineering process. All these documents are a vital part of good software development practice. Good documents are very useful and server the following purposes:

- Good documents enhance understandability and maintainability of a software product. They reduce the effort and time required for maintenance.
- Use documents help the users in effectively using the system.
- Good documents help in effectively handling the manpower turnover problem. Even when an engineer leaves the organization, and a new engineer comes in, he can build up the required knowledge easily.
- Production of good documents helps the manager in effectively tracking the progress of the project. The project manager knows that measurable progress is achieved if a piece of work is done and the required documents have been produced and reviewed.

Different types of software documents can broadly be classified into the following:

- Internal documentation
- External documentation

Internal documentation is the code comprehension features provided as part of the source code itself. Internal documentation is provided through appropriate module headers and comments embedded in the source code. Internal documentation is also provided through the useful variable names, module and function headers, code indentation, code structuring, use of enumerated types and constant

identifiers, use of user-defined data types, etc. Careful experiments suggest that out of all types of internal documentation meaningful variable names is most useful in understanding the code. This is of course in contrast to the common expectation that code commenting would be the most useful. The research finding is obviously true when comments are written without thought. For example, the following style of code commenting does not in any way help in understanding the code.

```
a = 10; /* a made 10 */
```

But even when code is carefully commented, meaningful variable names still are more helpful in understanding a piece of code. Good software development organizations usually ensure good internal documentation by appropriately formulating their coding standards and coding guidelines.

External documentation is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test documents, etc. A systematic software development style ensures that all these documents are produced in an orderly fashion.

10.1.5 Program Testing

Testing a program consists of providing the program with a set of test inputs (or test cases) and observing if the program behaves as expected. If the program fails to behave as expected, then the conditions under which failure occurs are noted for later debugging and correction.

Some commonly used terms associated with testing are:

- **Failure:** This is a manifestation of an error (or defect or bug). But, the mere presence of an error may not necessarily lead to a failure.
- **Test case:** This is the triplet [I,S,O], where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system.
- **Test suite:** This is the set of all test cases with which a given software product is to be tested.

10.1.5.1 Aim of testing

The aim of the testing process is to identify all defects existing in a software product. However, for most practical systems, even after satisfactorily carrying out the testing phase, it is not possible to guarantee that the software is error free. This is because of the fact that the input data domain of most software products is very large. It is not practical to test the software exhaustively with respect to each value that the input data may assume. Even with this practical limitation of the testing process, the importance of testing should not be underestimated. It must be remembered that testing does expose many defects existing in a software product. Thus, testing provides a practical way of reducing defects in a system and increasing the users' confidence in a developed system.

Differentiate between verification and validation.

Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase, whereas validation is the process of determining whether a fully developed system conforms to its requirements specification. Thus while verification is concerned with phase containment of errors, the aim of validation is that the final product be error free.

Design of test cases

Exhaustive testing of almost any non-trivial system is impractical due to the fact that the domain of input data values to most practical software systems is either extremely large or infinite. Therefore, we must design an optional test suite that is of reasonable size and can uncover as many errors existing in the system as possible. Actually, if test cases are selected randomly, many of these randomly selected test cases do not contribute to the significance of the test suite, i.e. they do not detect any additional defects not already being detected by other test cases in the suite. Thus, the number of random test cases in a test suite is, in general, not an indication of the effectiveness of the testing. In other words, testing a system using a large collection of test cases that are selected at random does not guarantee that all (or even most) of the errors in the system will be uncovered. Consider the following example code segment which finds the greater of two integer values x and y . This code segment has a simple programming error.

```
If (x>y) max = x;  
    else max = x;
```

For the above code segment, the test suite, $\{(x=3,y=2);(x=2,y=3)\}$ can detect the error, whereas a larger test suite $\{(x=3,y=2);(x=4,y=3);(x=5,y=1)\}$ does not detect the error. So, it would be incorrect to say that a larger test suite would always detect more errors than a smaller one, unless of course the larger test suite has also been carefully designed. This implies that the test suite should be carefully designed than picked randomly. Therefore, systematic approaches should be followed to design an optimal test suite. In an optimal test suite, each test case is designed to detect different errors.

Functional testing vs. Structural testing

In the black-box testing approach, test cases are designed using only the functional specification of the software, i.e. without any knowledge of the internal structure of the software. For this reason, black-box testing is known as functional testing. On the other hand, in the white-box testing approach, designing test cases requires thorough knowledge about the internal structure of software, and therefore the white-box testing is called structural testing.

10.2 BLACK-BOX TESTING

10.2.1 Testing in the large vs. testing in the small

Software products are normally tested first at the individual component (or unit) level. This is referred to as testing in the small. After testing all the components individually, the components are slowly integrated and tested at each level of integration (integration testing). Finally, the fully integrated system is tested (called system testing). Integration and system testing are known as testing in the large.

10.2.1.1 Unit testing

Unit testing is undertaken after a module has been coded and successfully reviewed. Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation.

In order to test a single module, a complete environment is needed to provide all that is necessary for execution of the module. That is, besides the module under test itself, the following steps are needed in order to be able to test the module:

- The procedures belonging to other modules that the module under test calls.
- Nonlocal data structures that the module accesses.
- A procedure to call the functions of the module under test with appropriate parameters.

Modules required to provide the necessary environment (which either call or are called by the module under test) is usually not available until they too have been unit tested, stubs and drivers are designed to provide the complete environment for a module. The role of stub and driver modules is pictorially shown in fig. 10.1. A stub procedure is a dummy procedure that has the same I/O parameters as the given procedure but has a highly simplified behavior. For example, a stub procedure may produce the expected behavior using a simple table lookup mechanism. A driver module contain the nonlocal data structures accessed by the module under test, and would also have the code to call the different functions of the module with appropriate parameter values.

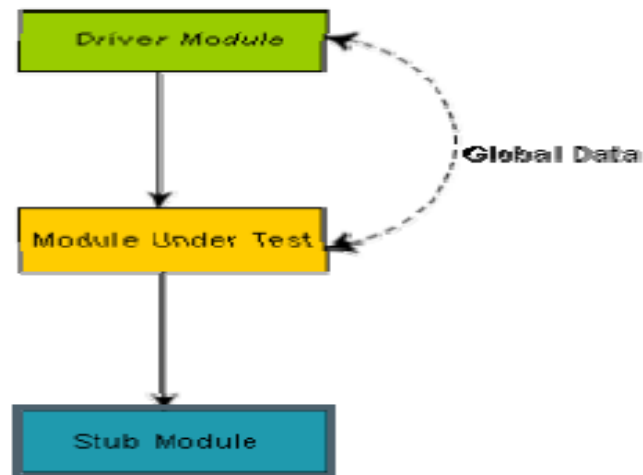


Fig. 10.1: Unit testing with the help of driver and stub modules

10.2.1.2 Black box testing

In the black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required. The following are the two main approaches to designing black box test cases.

- Equivalence class partitioning
- Boundary value analysis

10.2.2 Equivalence Class Partitioning

In this approach, the domain of input values to a program is partitioned into a set of equivalence classes. This partitioning is done such that the behavior of the program is similar for every input data belonging to the same equivalence class. The main idea behind defining the equivalence classes is that testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that equivalence class. Equivalence classes for software can be designed by examining the input data and output data. The following are some general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes should be defined.
2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for valid input values and another equivalence class for invalid input values should be defined.

Example#1: For a software that computes the square root of an input integer which can assume values in the range of 0 to 5000, there are three equivalence classes: The set of negative integers, the set of integers in the range of 0 and 5000, and the integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes and a possible test set can be: {-5,500,6000}.

Example#2: Design the black-box test suite for the following program. The program computes the intersection point of two straight lines and displays the result. It reads two integer pairs (m1, c1) and (m2, c2) defining the two straight lines of the form $y=mx + c$.

The equivalence classes are the following:

- Parallel lines ($m_1=m_2, c_1 \neq c_2$)
- Intersecting lines ($m_1 \neq m_2$)
- Coincident lines ($m_1=m_2, c_1=c_2$)

Now, selecting one representative value from each equivalence class, the test suit (2, 2) (2, 5), (5, 5) (7, 7), (10, 10) (10, 10) are obtained.

10.2.3 Boundary Value Analysis

A type of programming error frequently occurs at the boundaries of different equivalence classes of inputs. The reason behind such errors might purely be due to psychological factors. Programmers often fail to see the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use $<$ instead of \leq , or conversely \leq for $<$. Boundary value analysis leads to selection of test cases at the boundaries of the different equivalence classes.

Example: For a function that computes the square root of integer values in the range of 0 and 5000, the test cases must include the following values: {0, -1, 5000, 5001}.

Test cases for equivalence class testing and boundary value analysis for a problem
Let's consider a function that computes the square root of integer values in the range of 0 and 5000. For this particular problem, test cases corresponding to equivalence class testing and boundary value analysis have been found out earlier.

Check Your Progress I

Mark all options which are true.

1. The side effects of a function call include
 - a. modification of parameters passed by reference
 - b. modification of global variables
 - c. modification of I/O operations
 - d. all of the above
2. Code review for a module is carried out
 - a. as soon as skeletal code written
 - b. before the module is successfully compiled
 - c. after the module is successfully compiled and all the syntax errors have been eliminated
 - d. before the module is successfully compiled and all the syntax errors have been eliminated
3. An important factor that guides the integration plan for integration testing is
 - a. ER diagram

- b. data flow diagram
 - c. structure chart
 - d. none of the above
4. An integration testing approach, where all the modules making up a system are integrated in a single step is known as
- a. top-down integration testing
 - b. bottom-up integration testing
 - c. big-bang integration testing
 - d. mixed integration testing
5. An integration testing approach, where testing can start whenever modules become available is known as
- a. top-down integration testing
 - b. bottom-up integration testing
 - c. big-bang integration testing
 - d. mixed integration testing

Mark the following as either True or False. Justify your answer.

- 6. Coding standards are synonyms for coding guidelines.
- 7. During code inspection, you detect errors whereas during code testing you detect failures.
- 8. Out of all types of internal documentation (i.e. provided in the source code), careful commenting is most useful.
- 9. Error and failure are synonymous in software testing terminology.
- 10. Software verification and validation are synonyms terms.
- 11. The effectiveness of a test suite in detecting errors in a system can be determined by counting the number of test cases in the suite.

10.3 WHITE-BOX TESTING

One white-box testing strategy is said to be *stronger than* another strategy, if all types of errors detected by the first testing strategy is also detected by the second testing strategy, and the second testing strategy additionally detects some more types of errors. When two testing strategies detect errors that are different at least with respect to some types of errors, then they are called *complementary*. The

concepts of stronger and complementary testing are schematically illustrated in fig. 10.2.

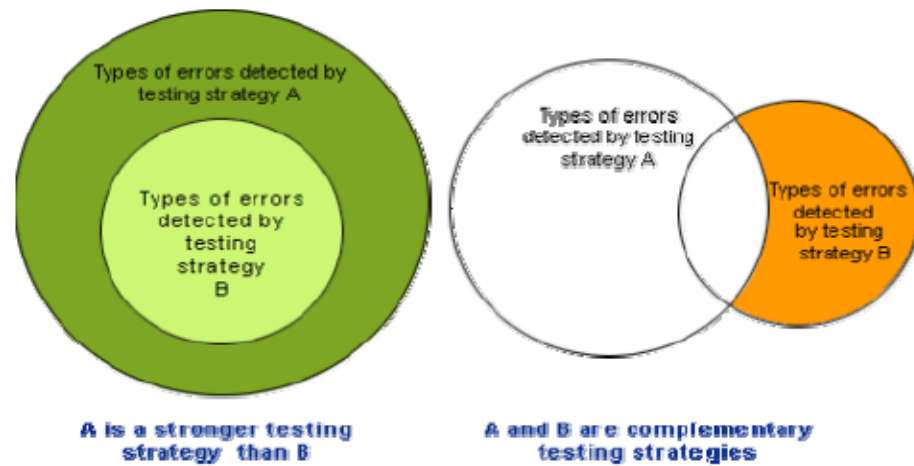


Fig. 10.2: Stronger and complementary testing strategies

10.3.1 Statement coverage

The statement coverage strategy aims to design test cases so that every statement in a program is executed at least once. The principal idea governing the statement coverage strategy is that unless a statement is executed, it is very hard to determine if an error exists in that statement. Unless a statement is executed, it is very difficult to observe whether it causes failure due to some illegal memory access, wrong result computation, etc. However, executing some statement once and observing that it behaves properly for that input value is no guarantee that it will behave correctly for all input values. In the following, designing of test cases using the statement coverage strategy have been shown.

Example: Consider the Euclid's GCD computation algorithm:

```
int compute_gcd(x, y)
int x, y;
{
    1 while (x != y){
    2 if (x > y) then
    3 x = x - y;
    4 else y = y - x;
    5 }
    6 return x;
```

}

By choosing the test set $\{(x=3, y=3), (x=4, y=3), (x=3, y=4)\}$, we can exercise the program such that all statements are executed at least once.

10.3.2 Branch coverage

In the branch coverage-based testing strategy, test cases are designed to make each branch condition to assume true and false values in turn. Branch testing is also known as edge testing as in this testing scheme, each edge of a program's control flow graph is traversed at least once.

It is obvious that branch testing guarantees statement coverage and thus is a stronger testing strategy compared to the statement coverage-based testing. For Euclid's GCD computation algorithm, the test cases for branch coverage can be $\{(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)\}$.

10.3.3 Condition coverage

In this structural testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, in the conditional expression $((c1.and.c2).or.c3)$, the components $c1$, $c2$ and $c3$ are each made to assume both true and false values. Branch testing is probably the simplest condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values. Thus, condition testing is a stronger testing strategy than branch testing and branch testing is stronger testing strategy than the statement coverage-based testing. For a composite conditional expression of n components, for condition coverage, 2^n test cases are required. Thus, for condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, a condition coverage-based testing technique is practical only if n (the number of conditions) is small.

10.3.4 Path coverage

The path coverage-based testing strategy requires us to design test cases such that all linearly independent paths in the program are executed at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.

10.3.5 Control Flow Graph (CFG)

A control flow graph describes the sequence in which the different instructions of a program get executed. In other words, a control flow graph describes how the control flows through the program. In order to draw the control flow graph of a program, all the statements of a program must be numbered first. The different numbered statements serve as nodes of the control flow graph (as shown in fig. 10.3). An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node.

The CFG for any program can be easily drawn by knowing how to represent the sequence, selection, and iteration type of statements in the CFG. After all, a program is made up from these types of statements. Fig. 10.3 summarizes how the CFG for these three types of statements can be drawn. It is important to note that for the iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore the control flow from the last statement of the loop is always to the top of the loop. Using these basic ideas, the CFG of Euclid's GCD computation algorithm can be drawn as shown in fig. 10.4.

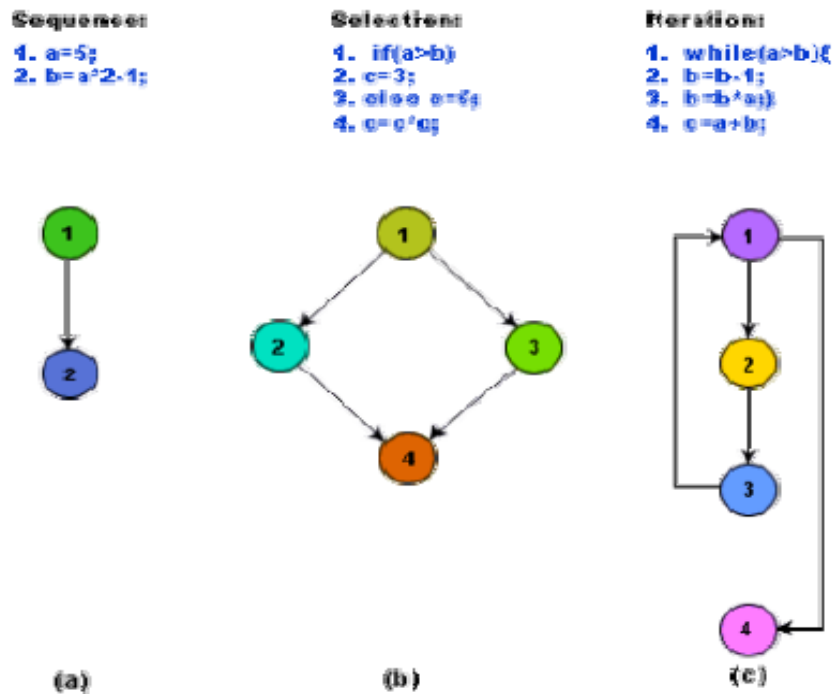
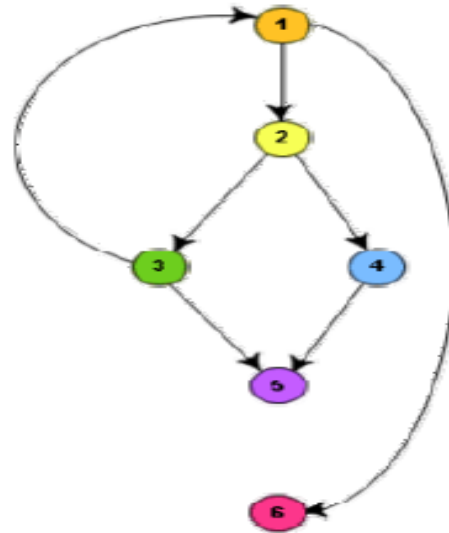


Fig. 10.3: CFG for (a) sequence, (b) selection, and (c) iteration type of constructs

```

int compute_gcd(int x, int y){
1 while(x!=y){
2   if(x>y) then
3     x=x-y;
4   else y=y-x;
5 }
6 return x;
}

```



(a) Example program

(b) Control Flow Graph

Fig. 10.4: Control flow diagram

Path

A path through a program is a node and edge sequence from the starting node to a terminal node of the control flow graph of a program. There can be more than one terminal node in a program. Writing test cases to cover all the paths of a typical program is impractical. For this reason, the path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths.

Linearly independent path

A linearly independent path is any path through the program that introduces at least one new edge that is not included in any other linearly independent paths. If a path has one new node compared to all other linearly independent paths, then the path is also linearly independent. This is because, any path having a new node automatically implies that it has a new edge. Thus, a path that is subpath of another path is not considered to be a linearly independent path.

Control flow graph

In order to understand the path coverage-based testing strategy, it is very much necessary to understand the control flow graph (CFG) of a program. Control flow graph (CFG) of a program has been discussed earlier.

Linearly independent path

The path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths. Linearly independent paths have been discussed earlier.

10.3.6 Cyclomatic complexity

For more complicated programs it is not easy to determine the number of independent paths of the program. McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program. Also, the McCabe's cyclomatic complexity is very simple to compute. Thus, the McCabe's cyclomatic complexity metric provides a practical way of determining the maximum number of linearly independent paths in a program. Though the McCabe's metric does not directly identify the linearly independent paths, but it informs approximately how many paths to look for.

There are three different ways to compute the cyclomatic complexity. The answers computed by the three methods are guaranteed to agree.

Method 1:

Given a control flow graph G of a program, the cyclomatic complexity $V(G)$ can be computed as: $V(G) = E - N + 2$

where N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph. For the CFG of example shown in fig. 10.4, $E=7$ and $N=6$. Therefore, the cyclomatic complexity $= 7-6+2 = 3$.

Method 2:

An alternative way of computing the cyclomatic complexity of a program from an inspection of its control flow graph is as follows:

$$V(G) = \text{Total number of bounded areas} + 1$$

In the program's control flow graph G , any region enclosed by nodes and edges can be called as a bounded area. This is an easy way to determine the McCabe's cyclomatic complexity. But, what if the graph G is not planar, i.e. however you draw the graph, two or more edges intersect? Actually, it can be shown that structured programs always yield planar graphs. But, presence of GOTO's can easily add intersecting edges. Therefore, for non-structured programs, this way of computing the McCabe's cyclomatic complexity cannot be used.

The number of bounded areas increases with the number of decision paths and loops. Therefore, the McCabe's metric provides a quantitative measure of testing difficulty and the ultimate reliability. For the CFG example shown in fig. 10.4, from a visual examination of the CFG the number of bounded areas is 2. Therefore the cyclomatic complexity, computed with this method is also $2+1 = 3$. This method provides a very easy way of computing the cyclomatic complexity of CFGs, just from a visual examination of the CFG. On the other hand, the other method of computing CFGs is more amenable to automation, i.e. it can be easily coded into a program which can be used to determine the cyclomatic complexities of arbitrary CFGs.

Method 3:

The cyclomatic complexity of a program can also be easily computed by computing the number of decision statements of the program. If N is the number of decision statement of a program, then the McCabe's metric is equal to $N+1$.

Data flow-based testing

Data flow-based testing method selects test paths of a program according to the locations of the definitions and uses of different variables in a program.

For a statement numbered S , let

DEF(S) = {X/statement S contains a definition of X}, and

USES(S) = {X/statement S contains a use of X}

For the statement **S:a=b+c;**, DEF(S) = {a}. USES(S) = {b,c}. The definition of variable X at statement S is said to be live at statement $S1$, if there exists a path from statement S to statement $S1$ which does not contain any definition of X .

The *definition-use chain* (or DU chain) of a variable X is of form $[X, S, S1]$, where S and $S1$ are statement numbers, such that $X \in \text{DEF}(S)$ and $X \in \text{USES}(S1)$, and the definition of X in the statement S is live at statement $S1$. One simple data flow testing strategy is to require that every DU chain be covered at least once. Data flow testing strategies are useful for selecting test paths of a program containing nested if and loop statements.

10.3.7 Mutation testing

In mutation testing, the software is first tested by using an initial test suite built up from the different white box testing strategies. After the initial testing is complete,

mutation testing is taken up. The idea behind mutation testing is to make few arbitrary changes to a program at a time. Each time the program is changed, it is called as a mutated program and the change effected is called as a mutant. A mutated program is tested against the full test suite of the program. If there exists at least one test case in the test suite for which a mutant gives an incorrect result, then the mutant is said to be dead. If a mutant remains alive even after all the test cases have been exhausted, the test data is enhanced to kill the mutant. The process of generation and killing of mutants can be automated by predefining a set of primitive changes that can be applied to the program. These primitive changes can be alterations such as changing an arithmetic operator, changing the value of a constant, changing a data type, etc. A major disadvantage of the mutation-based testing approach is that it is computationally very expensive, since a large number of possible mutants can be generated. Since mutation testing generates a large number of mutants and requires us to check each mutant with the full test suite, it is not suitable for manual testing. Mutation testing should be used in conjunction of some testing tool which would run all the test cases automatically.

10.4 DEBUGGING, INTEGRATION AND SYSTEM TESTING

10.4.1 Need for debugging

Once errors are identified in a program code, it is necessary to first identify the precise program statements responsible for the errors and then to fix them. Identifying errors in a program code and then fix them up are known as debugging.

10.4.2 Debugging approaches

The following are some of the approaches popularly adopted by programmers for debugging.

10.4.2.1 Brute Force Method

This is the most common method of debugging but is the least efficient method. In this approach, the program is loaded with print statements to print the intermediate

values with the hope that some of the printed values will help to identify the statement in error. This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly.

10.4.2.2 Backtracking

This is also a fairly common approach. In this approach, beginning from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered. Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large thus limiting the use of this approach.

10.4.2.3 Cause Elimination Method

In this approach, a list of causes which could possibly have contributed to the error symptom is developed and tests are conducted to eliminate each. A related technique of identification of the error from the error symptom is the software fault tree analysis.

10.4.2.4 Program Slicing

This technique is similar to back tracking. Here the search space is reduced by defining slices. A slice of a program for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable [Mund2002].

10.4.3 Debugging guidelines

Debugging is often carried out by programmers based on their ingenuity. The following are some general guidelines for effective debugging:

- Many times, debugging requires a thorough understanding of the program design. Trying to debug based on a partial understanding of the system design and implementation may require an inordinate amount of effort to be put into debugging even simple problems.

- Debugging may sometimes even require full redesign of the system. In such cases, a common mistake that novice programmers often make is attempting not to fix the error but its symptoms.
- One must be beware of the possibility that an error correction may introduce new errors. Therefore, after every round of error-fixing, regression testing must be carried out.

10.4.4 Program analysis tools

A program analysis tool means an automated tool that takes the source code or the executable code of a program as input and produces reports regarding several important characteristics of the program, such as its size, complexity, adequacy of commenting, adherence to programming standards, etc. We can classify these into two broad categories of program analysis tools:

- Static Analysis tools
- Dynamic Analysis tools

10.4.5 Static program analysis tools

Static analysis tool is also a program analysis tool. It assesses and computes various characteristics of a software product without executing it. Typically, static analysis tools analyze some structural representation of a program to arrive at certain analytical conclusions, e.g. that some structural properties hold. The structural properties that are usually analyzed are:

- Whether the coding standards have been adhered to?
- Certain programming errors such as uninitialized variables and mismatch between actual and formal parameters, variables that are declared but never used are also checked.

Code walk throughs and code inspections might be considered as static analysis methods. But, the term static program analysis is used to denote automated analysis tools. So, a compiler can be considered to be a static program analysis tool.

10.4.6 Dynamic program analysis tools

Dynamic program analysis techniques require the program to be executed and its actual behavior recorded. A dynamic analyzer usually instruments the code (i.e. adds additional statements in the source code to collect program execution traces). The instrumented code when executed allows us to record the behavior of the software for different test cases. After the software has been tested with its full test suite and its behavior recorded, the dynamic analysis tool carries out a post execution analysis and produces reports which describe the structural coverage that has been achieved by the complete test suite for the program. For example, the post execution dynamic analysis report might provide data on extent statement, branch and path coverage achieved.

Normally the dynamic analysis results are reported in the form of a histogram or a pie chart to describe the structural coverage achieved for different modules of the program. The output of a dynamic analysis tool can be stored and printed easily and provides evidence that thorough testing has been done. The dynamic analysis results the extent of testing performed in white-box mode. If the testing coverage is not satisfactory more test cases can be designed and added to the test suite. Further, dynamic analysis results can help to eliminate redundant test cases from the test suite.

10.4.7 Integration testing

The primary objective of integration testing is to test the module interfaces, i.e. there are no errors in the parameter passing, when one module invokes another module. During integration testing, different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the steps and the order in which modules are combined to realize the full system. After each integration step, the partially integrated system is tested. An important factor that guides the integration plan is the module dependency graph. The structure chart (or module dependency graph) denotes the order in which different modules call each other. By examining the structure chart, the integration plan can be developed.

10.4.8 Integration test approaches

There are four types of integration testing approaches. Any one (or a mixture) of the following approaches can be used to develop the integration test plan. Those approaches are the following:

- Big bang approach
- Top-down approach
- Bottom-up approach
- Mixed-approach

10.4.8.1 Big-Bang Integration Testing

It is the simplest integration testing approach, where all the modules making up a system are integrated in a single step. In simple words, all the modules of the system are simply put together and tested. However, this technique is practicable only for very small systems. The main problem with this approach is that once an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated. Therefore, debugging errors reported during big bang integration testing are very expensive to fix.

10.4.8.2 Bottom-Up Integration Testing

In bottom-up testing, each subsystem is tested separately and then the full system is tested. A subsystem might consist of many modules which communicate among each other through well-defined interfaces. The primary purpose of testing each subsystem is to test the interfaces among various modules making up the subsystem. Both control and data interfaces are tested. The test cases must be carefully chosen to exercise the interfaces in all possible manners.

Large software systems normally require several levels of subsystem testing; lower-level subsystems are successively combined to form higher-level subsystems. A principal advantage of bottom-up integration testing is that several disjoint subsystems can be tested simultaneously. In a pure bottom-up testing no stubs are required, only test-drivers are required. A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large

number of small subsystems. The extreme case corresponds to the big-bang approach.

10.4.8.3 Top-Down Integration Testing

Top-down integration testing starts with the main routine and one or two subordinate routines in the system. After the top-level ‘skeleton’ has been tested, the immediately subroutines of the ‘skeleton’ are combined with it and tested. Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test. A pure top-down integration does not require any driver routines. A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, many times it may become difficult to exercise the top-level routines in the desired manner since the lower-level routines perform several low-level functions such as I/O.

10.4.8.4 Mixed Integration Testing

A mixed (also called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are ready. The mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. In the mixed testing approaches, testing can start as and when modules become available. Therefore, this is one of the most commonly used integration testing approaches.

10.4.9 Phased vs. incremental testing

The different integration testing strategies are either phased or incremental. A comparison of these two strategies is as follows:

- In incremental integration testing, only one new module is added to the partial system each time.
- In phased integration, a group of related modules are added to the partial system each time.

Phased integration requires less number of integration steps compared to the incremental integration approach. However, when failures are detected, it is easier

to debug the system in the incremental testing approach since it is known that the error is caused by addition of a single module. In fact, big bang testing is a degenerate case of the phased integration testing approach.

10.4.10 System testing

System tests are designed to validate a fully developed system to assure that it meets its requirements. There are essentially three main kinds of system testing:

- **Alpha Testing.** Alpha testing refers to the system testing carried out by the test team within the developing organization.
- **Beta testing.** Beta testing is the system testing performed by a select group of friendly customers.
- **Acceptance Testing.** Acceptance testing is the system testing performed by the customer to determine whether he should accept the delivery of the system.

In each of the above types of tests, various kinds of test cases are designed by referring to the SRS document. Broadly, these tests can be classified into functionality and performance tests. The functionality tests test the functionality of the software to check whether it satisfies the functional requirements as documented in the SRS document. The performance tests test the conformance of the system with the nonfunctional requirements of the system.

10.4.11 Performance testing

Performance testing is carried out to check whether the system needs the non-functional requirements identified in the SRS document. There are several types of performance testing. Among of them nine types are discussed below. The types of performance testing to be carried out on a system depend on the different non-functional requirements of the system documented in the SRS document. All performance tests can be considered as black-box tests.

- Stress testing
- Volume testing
- Configuration testing
- Compatibility testing
- Regression testing

- Recovery testing
- Maintenance testing
- Documentation testing
- Usability testing

10.4.11.1 Stress Testing

Stress testing is also known as endurance testing. Stress testing evaluates system performance when it is stressed for short periods of time. Stress tests are black box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the capabilities of the software. Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity. For example, suppose an operating system is supposed to support 15 multiprogrammed jobs, the system is stressed by attempting to run 15 or more jobs simultaneously. A real-time system might be tested to determine the effect of simultaneous arrival of several high-priority interrupts.

Stress testing is especially important for systems that usually operate below the maximum capacity but are severely stressed at some peak demand hours. For example, if the non-functional requirement specification states that the response time should not be more than 20 secs per transaction when 60 concurrent users are working, then during the stress testing the response time is checked with 60 users working simultaneously.

10.4.11.2 Volume Testing

It is especially important to check whether the data structures (arrays, queues, stacks, etc.) have been designed to successfully extraordinary situations. For example, a compiler might be tested to check whether the symbol table overflows when a very large program is compiled.

10.4.11.3 Configuration Testing

This is used to analyze system behavior in various hardware and software configurations specified in the requirements. Sometimes systems are built in variable configurations for different users. For instance, we might define a minimal system to serve a single user, and other extension configurations to serve

additional users. The system is configured in each of the required configurations and it is checked if the system behaves correctly in all required configurations.

10.4.11.4 Compatibility Testing

This type of testing is required when the system interfaces with other types of systems. Compatibility aims to check whether the interface functions perform as required. For instance, if the system needs to communicate with a large database system to retrieve information, compatibility testing is required to test the speed and accuracy of data retrieval.

10.4.11.5 Regression Testing

This type of testing is required when the system being tested is an upgradation of an already existing system to fix some bugs or enhance functionality, performance, etc. Regression testing is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced due to the change or the bug fix. However, if only a few statements are changed, then the entire test suite need not be run - only those test cases that test the functions that are likely to be affected by the change need to be run.

10.4.11.6 Recovery Testing

Recovery testing tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc. The system is subjected to the loss of the mentioned resources (as applicable and discussed in the SRS document) and it is checked if the system recovers satisfactorily. For example, the printer can be disconnected to check if the system hangs. Or, the power may be shut down to check the extent of data loss and corruption.

10.4.11.7 Maintenance Testing

This testing addresses the diagnostic programs, and other procedures that are required to be developed to help maintenance of the system. It is verified that the artifacts exist and they perform properly.

10.4.11.8 Documentation Testing

It is checked that the required user manual, maintenance manuals, and technical manuals exist and are consistent. If the requirements specify the types of audience for which a specific manual should be designed, then the manual is checked for compliance.

10.4.11.9 Usability Testing

Usability testing concerns checking the user interface to see if it meets all user requirements concerning the user interface. During usability testing, the display screens, report formats, and other aspects relating to the user interface requirements are tested.

10.4.11.10 Error seeding

Sometimes the customer might specify the maximum number of allowable errors that may be present in the delivered system. These are often expressed in terms of maximum number of allowable errors per line of source code. Error seed can be used to estimate the number of residual errors in a system.

Error seeding, as the name implies, seeds the code with some known errors. In other words, some artificial errors are introduced into the program artificially. The number of these seeded errors detected in the course of the standard testing procedure is determined. These values in conjunction with the number of unseeded errors detected can be used to predict:

- The number of errors remaining in the product.
- The effectiveness of the testing strategy.

Let N be the total number of defects in the system and let n of these defects be found by testing.

Let S be the total number of seeded defects, and let s of these defects be found during testing.

$$n/N = s/S$$

or

$$N = S \times n/s$$

$$\text{Defects still remaining after testing} = N - n = n \times (S - s)/s$$

Error seeding works satisfactorily only if the kind of seeded errors matches closely with the kind of defects that actually exist. However, it is difficult to predict the

types of errors that exist in software. To some extent, the different categories of errors that remain can be estimated to a first approximation by analyzing historical data of similar projects. Due to the shortcoming that the types of seeded errors should match closely with the types of errors actually existing in the code, error seeding is useful only to a moderate extent.

10.4.11.11 Regression testing

Regression testing does not belong to either unit test, integration test, or system testing. Instead, it is a separate dimension to these three forms of testing. The functionality of regression testing has been discussed earlier.

Check Your Progress II

1. When a system interfaces with other types of systems then that time the testing that will be required is
 - a. volume testing
 - b. configuration testing
 - c. compatibility testing
 - d. maintenance testing
2. When a system being tested is an upgradation of an already existing system to fix some bugs or enhance functionality, performance, etc. then the testing required to be performed is:
 - a. documentation testing
 - b. regression testing
 - c. maintenance testing
 - d. recovery testing
3. Error seed can be used
 - a. to estimate the total number of defects in the system
 - b. to estimate the total number of seeded defects in a system
 - c. to estimate the number of residual errors in a system
 - d. none of the above
4. Test SUMMARY report comprises of
 - a. the total number of tests that have been applied to a subsystem
 - b. how many tests have been successful

- c. how many tests have been unsuccessful
- d. all of the above

Mark the following as either True or False. Justify your answer.

5. The number of test cases required for statement coverage-based testing of a program can be greater than those required for path coverage-based testing of the same program.
6. Condition testing strategy is a stronger testing strategy than branch testing strategy.
7. A program can have more than one linearly independent path.
8. Once the McCabe's Cyclomatic complexity of a program has been determined, it is very easy to identify all the linearly independent paths of the program.
9. Introduction of additional edges and nodes in the CFG due to introduction of sequence types of statements in the program can increase the cyclomatic complexity of the program.
10. A pure top-down integration testing does not require the use of any stub modules.
11. Adherence to coding standards is checked during the system testing stage.
12. Development of suitable driver and stub functions are essential for carrying out effective system testing of a product.
13. System testing can be considered as a white box testing.
14. The main purpose of integration testing is to find design errors.

10.5 MODEL QUESTIONS

1. What are the different ways of documenting program code? Which of these is usually the most useful while understanding a piece of code?
2. What is a coding standard? Identify the problems that might occur if the engineers of an organization do not adhere to any coding standard.
3. What is the difference between coding standards and coding guidelines? Why are these considered as important in a software development organization?
4. Write down five important coding standards.
5. Write down five important coding guidelines.

6. What do you mean by side effects of a function call? Why are obscure side effects undesirable?
7. What is meant by code review? Why is it required to be completed before performing integration and system testing?
8. Identify the type of errors that can be detected during code walk throughs.
9. Identify the type of errors that can be detected during code inspection.
10. What is clean room testing?
11. Why is it important to properly document a software product?
12. Differentiate between the external and internal documentation of a software product.
13. Identify the necessity of testing of a software product.
14. Distinguish between error and failure. Testing detects which of these two? Justify it.
15. Differentiate between verification and validation in the context of software testing.
16. Is random selection of test cases effective? Justify.
17. Write down major differences between functional testing and structural testing.
18. Do you agree with the statement: “The effectiveness of a testing suite in detecting errors in a system can be determined by examining the number of test cases in the suite”. Justify your answer.
19. What are driver and stub modules in the context of unit testing of a software product?
20. Given a software and its requirements specification document, how can black-box test suites for this software be designed?
21. Identify two guidelines for the design of equivalence classes for a problem.
22. Explain why boundary value analysis is so important for the design of black box test suite for a problem.
23. Compare the features of stronger testing with the features of complementary testing.
24. Which is strongest structural testing technique among statement coverage-based testing, branch coverage-based testing, and condition coverage-based testing? Why?

25. Discuss how does control flow graph (CFG) of a problem help in understanding of path coverage based testing strategy.
26. Draw the control flow graph for the following function named find-maximum. From the control flow graph, determines its Cyclomatic complexity.

```
int find-maximum(int i, int j, int k) { int max; if(i>j) then if(i>k) then max = i; else max = k; else if(j>k) max = j; else max = k; return(max); }
```

27. What is the difference between path and linearly independent path in terms of control flow graph (CFG) of a problem?
28. Define a metric form which the upper bound for the number of linearly independent paths of a program can be computed.
29. Consider the following C function named bin-search:

```
/* num is the number the function searches in a presorted integer array arr */  
int bin_search(int num) { int min, max; min = 0; max = 100;  
while(min!=max){ if (arr[(min+max)/2]>num) max = (min+max)/2; else  
if(arr[(min+max)/2]<num) min = (min+max)/2; else return((min+max)/2); }  
return(-1); }
```

Determine the cyclomatic complexity of the above problem.

30. What is meant by data flow-based testing approach?
31. What are the advantages of performing mutation testing upon a software product?
32. Write down three general guidelines for performing effective debugging.
33. Distinguish between the static and dynamic analysis of a program. How are static and dynamic program analysis results useful?
34. What do you understand by the term integration testing? What are the different types of integration testing methods that can be used to carry out integration testing of a large software product?
35. Do you agree with the following statement: "System testing can be considered as a pure black-box test." Justify your answer.

36. What do you understand by performance testing? Write down the different types of performance testing.
37. What is meant by error seeding?
38. Explain the necessity of performing regression testing.

UNIT XI: SOFTWARE PROJECT PLANNING

11.0 LEARNING OBJECTIVES

At the end of this lesson the student would be able to:

- Identify the job responsibilities of a software project manager.
- Identify the necessary skills required in order to perform software project management.
- Identify the essential activities of project planning.
- Determine the different project related estimates performed by a project manager and suitably order those estimates.
- Explain what is meant by Sliding Window Planning.
- Explain what is Software Project Management Plan (SPMP).
- Identify and explain two metrics for software project size estimation.
- Identify the shortcomings of function point (FP) metric.
- Explain the necessity of feature point metric in the context of project size estimation.
- Identify the types of project-parameter estimation technique.
- Differentiate among organic, semidetached and embedded software projects.
- Explain basic COCOMO.
- Differentiate between basic COCOMO model and intermediate COCOMO model.
- Explain the complete COCOMO model.
- Identify why careful planning of staffing pattern for a project is so important.
- Determine numerically how change of project duration affects the overall effort and cost.
- Identify five necessary tasks taken by a project manager in order to perform project scheduling.
- Explain the usefulness of work breakdown structure.

- Explain activity networks and critical path method.
- Develop the Gantt chart for a project.
- Develop PERT chart for a project.

11.1 Project Planning and Project Estimation Techniques

11.1.1 Responsibilities of a software project manager

Software project managers take the overall responsibility of steering a project to success. It is very difficult to objectively describe the job responsibilities of a project manager. The job responsibility of a project manager ranges from invisible activities like building up team morale to highly visible customer presentations. Most managers take responsibility for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, interfacing with clients, managerial report writing and presentations, etc. These activities are certainly numerous, varied and difficult to enumerate, but these activities can be broadly classified into project planning, and project monitoring and control activities. The project planning activity is undertaken before the development starts to plan the activities to be undertaken during development. The project monitoring and control activities are undertaken once the development activities start with the aim of ensuring that the development proceeds as per plan and changing the plan whenever required to cope up with the situation.

Skills necessary for software project management

A theoretical knowledge of different project management techniques is certainly necessary to become a successful project manager. However, effective software project management frequently calls for good qualitative judgment and decision taking capabilities. In addition to having a good grasp of the latest software project management techniques such as cost estimation, risk management, configuration management, project managers need good communication skills and the ability get work done. However, some skills such as tracking and controlling the progress of the project, customer interaction, managerial presentations, and team building are

largely acquired through experience. None the less, the importance of sound knowledge of the prevalent project management techniques cannot be overemphasized.

11.1.2 Project planning

Once a project is found to be feasible, software project managers undertake project planning. Project planning is undertaken and completed even before any development activity starts. Project planning consists of the following essential activities:

- Estimating the following attributes of the project:

Project size: What will be problem complexity in terms of the effort and time required to develop the product?

Cost: How much is it going to cost to develop the project?

Duration: How long is it going to take to complete development?

Effort: How much effort would be required?

The effectiveness of the subsequent planning activities is based on the accuracy of these estimations.

- Scheduling manpower and other resources
- Staff organization and staffing plans
- Risk identification, analysis, and abatement planning
- Miscellaneous plans such as quality assurance plan, configuration management plan, etc.

11.1.3 Precedence ordering among project planning activities

Different project related estimates done by a project manager have already been discussed. Fig. 11.1 shows the order in which important project planning activities may be undertaken. From fig. 11.1 it can be easily observed that size estimation is the first activity. It is also the most fundamental parameter based on which all other planning activities are carried out. Other estimations such as estimation of effort, cost, resource, and project duration are also very important components of project planning.

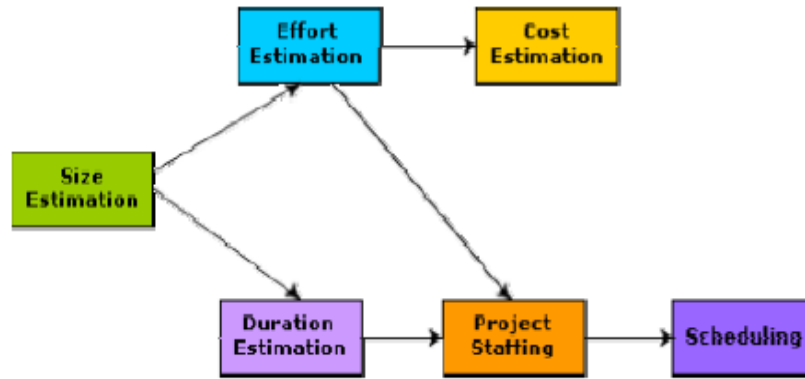


Fig. 11.1: Precedence ordering among planning activities

11.1.4 Sliding Window Planning

Project planning requires utmost care and attention since commitment to unrealistic time and resource estimates result in schedule slippage. Schedule delays can cause customer dissatisfaction and adversely affect team morale. It can even cause project failure. However, project planning is a very challenging activity. Especially for large projects, it is very much difficult to make accurate plans. A part of this difficulty is due to the fact that the proper parameters, scope of the project, project staff, etc. may change during the span of the project. In order to overcome this problem, sometimes project managers undertake project planning in stages. Planning a project over a number of stages protects managers from making big commitments too early. This technique of staggered planning is known as Sliding Window Planning. In the sliding window technique, starting with an initial plan, the project is planned more accurately in successive development stages. At the start of a project, project managers have incomplete knowledge about the details of the project. Their information base gradually improves as the project progresses through different phases. After the completion of every phase, the project managers can plan each subsequent phase more accurately and with increasing levels of confidence.

11.1.5 Software Project Management Plan (SPMP)

Once project planning is complete, project managers document their plans in a Software Project Management Plan (SPMP) document. The SPMP document

should discuss a list of different items that have been discussed below. This list can be used as a possible organization of the SPMP document. Organization of the Software Project Management Plan (SPMP) Document

1. Introduction

- (a) Objectives
- (b) Major Functions
- (c) Performance Issues
- (d) Management and Technical Constraints

2. Project Estimates

- (a) Historical Data Used
- (b) Estimation Techniques Used
- (c) Effort, Resource, Cost, and Project Duration Estimates

3. Schedule

- (a) Work Breakdown Structure
- (b) Task Network Representation
- (c) Gantt Chart Representation
- (d) PERT Chart Representation

4. Project Resources

- (a) People
- (b) Hardware and Software
- (c) Special Resources

5. Staff Organization

- (a) Team Structure
- (b) Management Reporting

6. Risk Management Plan

- (a) Risk Analysis
- (b) Risk Identification

- (c) Risk Estimation
- (d) Risk Abatement Procedures

7. Project Tracking and Control Plan

8. Miscellaneous Plans

- (a) Process Tailoring
- (b) Quality Assurance Plan
- (c) Configuration Management Plan
- (d) Validation and Verification
- (e) System Testing Plan
- (f) Delivery, Installation, and Maintenance Plan

11.1.6 Metrics for software project size estimation

Accurate estimation of the problem size is fundamental to satisfactory estimation of effort, time duration and cost of a software project. In order to be able to accurately estimate the project size, some important metrics should be defined in terms of which the project size can be expressed. The size of a problem is obviously not the number of bytes that the source code occupies. It is neither the byte size of the executable code. The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.

Currently two metrics are popularly being used widely to estimate size: lines of code (LOC) and function point (FP). The usage of each of these metrics in project size estimation has its own advantages and disadvantages.

11.1.6.1 Lines of Code (LOC)

LOC is the simplest among all metrics available to estimate project size. This metric is very popular because it is the simplest to use. Using this metric, the project size is estimated by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, lines used for commenting the code and the header lines should be ignored.

Determining the LOC count at the end of a project is a very simple job. However, accurate estimation of the LOC count at the beginning of a project is very difficult.

In order to estimate the LOC count at the beginning of a project, project managers usually divide the problem into modules, and each module into submodules and so on, until the sizes of the different leaf-level modules can be approximately predicted. To be able to do this, past experience in developing similar products is helpful. By using the estimation of the lowest level modules, project managers arrive at the total size estimation.

11.1.6.2 Function point (FP)

Function point metric was proposed by Albrecht [1983]. This metric overcomes many of the shortcomings of the LOC metric. Since its inception in late 1970s, function point metric has been slowly gaining popularity. One of the important advantages of using the function point metric is that it can be used to easily estimate the size of a software product directly from the problem specification. This is in contrast to the LOC metric, where the size can be accurately determined only after the product has fully been developed. The conceptual idea behind the function point metric is that the size of a software product is directly dependent on the number of different functions or features it supports. A software product supporting many features would certainly be of larger size than a product with less number of features. Each function when invoked reads some input data and transforms it to the corresponding output data. For example, the issue book feature (as shown in fig. 11.2) of a Library Automation Software takes the name of the book as input and displays its location and the number of copies available. Thus, a computation of the number of input and the output data values to a system gives some indication of the number of functions supported by the system. Albrecht postulated that in addition to the number of basic functions that a software performs, the size is also dependent on the number of files and the number of interfaces.

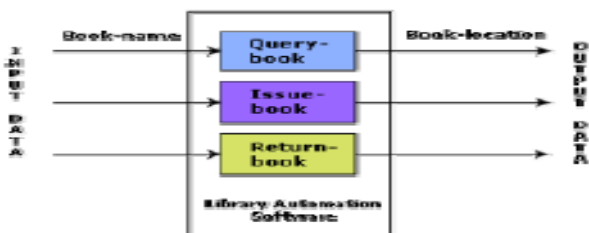


Fig. 11.2: System function as a map of input data to output data

Besides using the number of input and output data values, function point metric computes the size of a software product (in units of functions points or FPs) using three other characteristics of the product as shown in the following expression. The size of a product in function points (FP) can be expressed as the weighted sum of these five problem characteristics. The weights associated with the five characteristics were proposed empirically and validated by the observations over many projects. Function point is computed in two steps. The first step is to compute the unadjusted function point (UFP).

$$\text{UFP} = (\text{Number of inputs}) * 4 + (\text{Number of outputs}) * 5 + (\text{Number of inquiries}) * 4 + (\text{Number of files}) * 10 + (\text{Number of interfaces}) * 10$$

Number of inputs: Each data item input by the user is counted. Data inputs should be distinguished from user inquiries. Inquiries are user commands such as print-account-balance. Inquiries are counted separately. It must be noted that individual data items input by the user are not considered in the calculation of the number of inputs, but a group of related inputs are considered as a single input.

For example, while entering the data concerning an employee to an employee payroll software; the data items name, age, sex, address, phone number, etc. are together considered as a single input. All these data items can be considered to be related, since they pertain to a single employee.

Number of outputs: The outputs considered refer to reports printed, screen outputs, error messages produced, etc. While outputting the number of outputs the individual data items within a report are not considered, but a set of related data items is counted as one input.

Number of inquiries: Number of inquiries is the number of distinct interactive queries which can be made by the users. These inquiries are the user commands which require specific action by the system.

Number of files: Each logical file is counted. A logical file means groups of logically related data. Thus, logical files can be data structures or physical files.

Number of interfaces: Here the interfaces considered are the interfaces used to exchange information with other systems. Examples of such interfaces are data files on tapes, disks, communication links with other systems etc.

Once the unadjusted function point (UFP) is computed, the technical complexity factor (TCF) is computed next. TCF refines the UFP measure by considering fourteen other factors such as high transaction rates, throughput, and response time requirements, etc. Each of these 14 factors is assigned from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). Now, TCF is computed as $(0.65+0.01*DI)$. As DI can vary from 0 to 70, TCF can vary from 0.65 to 1.35. Finally, $FP=UFP*TCF$.

Shortcomings of function point (FP) metric

LOC as a measure of problem size has several shortcomings:

- LOC gives a numerical value of problem size that can vary widely with individual coding style – different programmers lay out their code in different ways. For example, one programmer might write several source instructions on a single line whereas another might split a single instruction across several lines. Of course, this problem can be easily overcome by counting the language tokens in the program rather than the lines of code. However, a more intricate problem arises because the length of a program depends on the choice of instructions used in writing the program. Therefore, even for the same problem, different programmers might come up with programs having different LOC counts. This situation does not improve even if language tokens are counted instead of lines of code.
- A good problem size measure should consider the overall complexity of the problem and the effort needed to solve it. That is, it should consider the local effort needed to specify, design, code, test, etc. and not just the coding effort. LOC, however, focuses on the coding activity alone; it merely computes the number of source lines in the final program. We have already seen that coding is only a small part of the overall software development activities. It is also wrong to argue that the overall product development effort is proportional to the effort required in writing the program code. This is because even though the design might be very complex, the code might be straightforward and vice versa. In such cases, code size is a grossly improper indicator of the problem size.

- LOC measure correlates poorly with the quality and efficiency of the code. Larger code size does not necessarily imply better quality or higher efficiency. Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set. In fact, it is very likely that a poor and sloppily written piece of code might have larger number of source instructions than a piece that is neat and efficient.
- LOC metric penalizes use of higher-level programming languages, code reuse, etc. The paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower. This would show up as smaller program size. Thus, if managers use the LOC count as a measure of the effort put in the different engineers (that is, productivity), they would be discouraging code reuse by engineers.
- LOC metric measures the lexical complexity of a program and does not address the more important but subtle issues of logical or structural complexities. Between two programs with equal LOC count, a program having complex logic would require much more effort to develop than a program with very simple logic. To realize why this is so, consider the effort required to develop a program having multiple nested loop and decision constructs with another program having only sequential control flow.
- It is very difficult to accurately estimate LOC in the final product from the problem specification. The LOC count can be accurately computed only after the code has been fully developed. Therefore, the LOC metric is little use to the project managers during project planning, since project planning is carried out even before any development activity has started. This possibly is the biggest shortcoming of the LOC metric from the project manager's perspective.

11.1.6.3 Feature point metric

A major shortcoming of the function point measure is that it does not take into account the algorithmic complexity of software. That is, the function point metric implicitly assumes that the effort required to design and develop any two functionalities of the system is the same. But, we know that this is normally not

true, the effort required to develop any two functionalities may vary widely. It only takes the number of functions that the system supports into consideration without distinguishing the difficulty level of developing the various functionalities. To overcome this problem, an extension of the function point metric called feature point metric is proposed.

Feature point metric incorporates an extra parameter algorithm complexity. This parameter ensures that the computed size using the feature point metric reflects the fact that the more is the complexity of a function, the greater is the effort required to develop it and therefore its size should be larger compared to simpler functions.

11.1.7 Project Estimation techniques

Estimation of various project parameters is a basic project planning activity. The important project parameters that are estimated include: project size, effort required to develop the software, project duration, and cost. These estimates not only help in quoting the project cost to the customer, but are also useful in resource planning and scheduling. There are three broad categories of estimation techniques:

- Empirical estimation techniques
- Heuristic techniques
- Analytical estimation techniques

11.1.7.1 Empirical Estimation Techniques

Empirical estimation techniques are based on making an educated guess of the project parameters. While using this technique, prior experience with development of similar products is helpful. Although empirical estimation techniques are based on common sense, different activities involved in estimation have been formalized over the years. Two popular empirical estimation techniques are: Expert judgment technique and Delphi cost estimation.

Expert Judgment Technique

Expert judgment is one of the most widely used estimation techniques. In this approach, an expert makes an educated guess of the problem size after analyzing the problem thoroughly. Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) of the system and then combines them to

arrive at the overall estimate. However, this technique is subject to human errors and individual bias. Also, it is possible that the expert may overlook some factors inadvertently. Further, an expert making an estimate may not have experience and knowledge of all aspects of a project. For example, he may be conversant with the database and user interface parts but may not be very knowledgeable about the computer communication part.

A more refined form of expert judgment is the estimation made by group of experts. Estimation by a group of experts minimizes factors such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates. However, the estimate made by a group of experts may still exhibit bias on issues where the entire group of experts may be biased due to reasons such as political considerations. Also, the decision made by the group may be dominated by overly assertive members.

Delphi cost estimation

Delphi cost estimation approach tries to overcome some of the shortcomings of the expert judgment approach. Delphi estimation is carried out by a team comprising of a group of experts and a coordinator. In this approach, the coordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit to the coordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced his estimation. The coordinator prepares and distributes the SUMMARY of the responses of all the estimators, and includes any unusual rationale noted by any of the estimators. Based on this SUMMARY, the estimators re-estimate. This process is iterated for several rounds. However, no discussion among the estimators is allowed during the entire estimation process. The idea behind this is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior. After the completion of several iterations of estimations, the coordinator takes the responsibility of compiling the results and preparing the final estimate.

11.1.7.2 Heuristic Techniques

Heuristic techniques assume that the relationships among the different project parameters can be modeled using suitable mathematical expressions. Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the value of the basic parameters in the mathematical expression. Different heuristic estimation models can be divided into the following two classes: single variable model and the multi variable model. Single variable estimation models provide a means to estimate the desired characteristics of a problem, using some previously estimated basic (independent) characteristic of the software product such as its size. A single variable estimation model takes the following form:

$$\text{Estimated Parameter} = c_1 * e^{d_1}$$

In the above expression, e is the characteristic of the software which has already been estimated (independent variable). *Estimated Parameter* is the dependent parameter to be estimated. The dependent parameter to be estimated could be effort, project duration, staff size, etc. c_1 and d_1 are constants. The values of the constants c_1 and d_1 are usually determined using data collected from past projects (historical data). The basic COCOMO model is an example of single variable cost estimation model.

A multivariable cost estimation model takes the following form:

$$\text{Estimated Resource} = c_1 * e_1^{d_1} + c_2 * e_2^{d_2} + \dots$$

Where e_1, e_2, \dots are the basic (independent) characteristics of the software already estimated, and $c_1, c_2, d_1, d_2, \dots$ are constants. Multivariable estimation models are expected to give more accurate estimates compared to the single variable models, since a project parameter is typically influenced by several independent parameters. The independent parameters influence the dependent parameter to different extents. This is modeled by the constants $c_1, c_2, d_1, d_2, \dots$. Values of these constants are usually determined from historical data. The intermediate COCOMO model can be considered to be an example of a multivariable estimation model.

11.1.7.3 Analytical Estimation Techniques

Analytical estimation techniques derive the required results starting with basic assumptions regarding the project. Thus, unlike empirical and heuristic techniques, analytical techniques do have scientific basis. Halstead's software science is an example of an analytical technique. Halstead's software science can be used to derive some interesting results starting with a few simple assumptions. Halstead's software science is especially useful for estimating software maintenance efforts. In fact, it outperforms both empirical and heuristic techniques when used for predicting software maintenance efforts.

Halstead's Software Science – An Analytical Technique

Halstead's software science is an analytical technique to measure size, development effort, and development cost of software products. Halstead used a few primitive program parameters to develop the expressions for over all program length, potential minimum value, actual volume, effort, and development time.

For a given program, let:

- η_1 be the number of unique operators used in the program,
- η_2 be the number of unique operands used in the program,
- N_1 be the total number of operators used in the program,
- N_2 be the total number of operands used in the program.

Length and Vocabulary

The length of a program as defined by Halstead, quantifies total usage of all operators and operands in the program. Thus, length $N = N_1 + N_2$. Halstead's definition of the length of the program as the total number of operators and operands roughly agrees with the intuitive notation of the program length as the total number of tokens used in the program.

The program vocabulary is the number of unique operators and operands used in the program. Thus, *program vocabulary* $\eta = \eta_1 + \eta_2$.

Program Volume

The length of a program (i.e. the total number of operators and operands used in the code) depends on the choice of the operators and operands used. In other words, for the same programming problem, the length would depend on the programming style. This type of dependency would produce different measures of

length for essentially the same problem when different programming languages are used. Thus, while expressing program size, the programming language used must be taken into consideration:

$$V = N \log_2 \eta$$

Here the program volume V is the minimum number of bits needed to encode the program. In fact, to represent η different identifiers uniquely, at least $\log_2 \eta$ bits (where η is the program vocabulary) will be needed. In this scheme, $N \log_2 \eta$ bits will be needed to store a program of length N . Therefore, the volume V represents the size of the program by approximately compensating for the effect of the programming language used.

Potential Minimum Volume

The potential minimum volume V^* is defined as the volume of most succinct program in which a problem can be coded. The minimum volume is obtained when the program can be expressed using a single source code instruction say a function call like $\text{foo}(\)$; In other words, the volume is bound from below due to the fact that a program would have at least two operators and no less than the requisite number of operands.

Thus, if an algorithm operates on input and output data d_1, d_2, \dots, d_n , the most succinct program would be $f(d_1, d_2, \dots, d_n)$; for which $\eta_1 = 2, \eta_2 = n$. Therefore, $V^* = (2 + \eta_2) \log_2 (2 + \eta_2)$.

The program level L is given by $L = V^*/V$. The concept of program level L is introduced in an attempt to measure the level of abstraction provided by the programming language. Using this definition, languages can be ranked into levels that also appear intuitively correct.

The above result implies that the higher the level of a language, the less effort it takes to develop a program using that language. This result agrees with the intuitive notion that it takes more effort to develop a program in assembly language than to develop a program in a high-level language to solve a problem.

Effort and Time

The effort required to develop a program can be obtained by dividing the program volume with the level of the programming language used to develop the code. Thus, effort $E = V/L$, where E is the number of mental discriminations required to implement the program and also the effort required to read and understand the

program. Thus, the programming effort $E = V^2/V^*$ (since $L = V^*/V$) varies as the square of the volume. Experience shows that E is well correlated to the effort needed for maintenance of an existing program.

The programmer's time $T = E/S$, where S the speed of mental discriminations. The value of S has been empirically developed from psychological reasoning, and its recommended value for programming applications is 18.

Length Estimation

Even though the length of a program can be found by calculating the total number of operators and operands in a program, Halstead suggests a way to determine the length of a program using the number of unique operators and operands used in the program. Using this method, the program parameters such as length, volume, cost, effort, etc. can be determined even before the start of any programming activity. His method is summarized below.

Halstead assumed that it is quite unlikely that a program has several identical parts – in formal language terminology identical substrings – of length greater than η (η being the program vocabulary). In fact, once a piece of code occurs identically at several places, it is made into a procedure or a function. Thus, it can be assumed that any program of length N consists of N/η unique strings of length η . Now, it is standard combinatorial result that for any given alphabet of size K , there are exactly K^r different strings of length r .

Thus,

$$N/\eta \leq \eta^n \text{ Or, } N \leq \eta^{n+1}$$

Since operators and operands usually alternate in a program, the upper bound can be further refined into $N \leq \eta \eta_1^{n_1} \eta_2^{n_2}$. Also, N must include not only the ordered set of n elements, but it should also include all possible subsets of that ordered sets, i.e. the power set of N strings (This particular reasoning of Halstead is not very convincing!!!).

Therefore,

$$2^N = \eta \eta_1^{n_1} \eta_2^{n_2}$$

Or, taking logarithm on both sides,

$$N = \log_2 \eta + \log_2 (\eta_1^{n_1} \eta_2^{n_2})$$

So we get,

$$N = \log_2 (\eta \eta_1^{n_1} \eta_2^{n_2})$$

(approximately, by ignoring $\log_2 \eta$)

Or,

$$N = \log_2 \eta_1^{\eta_1} + \log_2 \eta_2^{\eta_2}$$

$$= \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

Experimental evidence gathered from the analysis of larger number of programs suggests that the computed and actual lengths match very closely. However, the results may be inaccurate when small programs when considered individually.

In conclusion, Halstead's theory tries to provide a formal definition and quantification of such qualitative attributes as program complexity, ease of understanding, and the level of abstraction based on some low-level parameters such as the number of operands, and operators appearing in the program. Halstead's software science provides gross estimation of properties of a large collection of software, but extends to individual cases rather inaccurately.

Example:

Let us consider the following C program:

```
main( )  
{  
int a, b, c, avg;  
scanf(“%d %d %d”, &a, &b, &c);  
avg = (a+b+c)/3;  
printf(“avg = %d”, avg);  
}
```

The unique operators are:

main(), {}, int, scanf, &, “,” “;”, =, +, /, printf

The unique operands are:

a, b, c, &a, &b, &c, a+b+c, avg, 3,
“%d %d %d”, “avg = %d”

Therefore,

$$\eta_1 = 12, \eta_2 = 11$$

$$\text{Estimated Length} = (12 * \log 12 + 11 * \log 11)$$

$$= (12 * 3.58 + 11 * 3.45)$$

$$= (43 + 38) = 81$$

$$\text{Volume} = \text{Length} * \log(23)$$

= 81*4.52

= 366

11.2 COCOMO MODEL

11.2.1 Organic, Semidetached and Embedded software projects

Boehm postulated that any software development project can be classified into one of the following three categories based on the development complexity: organic, semidetached, and embedded. In order to classify a product into the identified categories, Boehm not only considered the characteristics of the product but also those of the development team and development environment. Roughly speaking, these three product classes correspond to application, utility and system programs, respectively. Normally, data processing programs are considered to be application programs. Compilers, linkers, etc., are utility programs. Operating systems and real-time system programs, etc. are system programs. System programs interact directly with the hardware and typically involve meeting timing constraints and concurrent processing.

Boehm's [1981] definition of organic, semidetached, and embedded systems are elaborated below.

Organic: A development project can be considered of organic type, if the project deals with developing a well understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.

Semidetached: A development project can be considered of semidetached type, if the development consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

Embedded: A development project is considered to be of embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational procedures exist.

11.2.2 COCOMO

COCOMO (Constructive Cost Estimation Model) was proposed by Boehm [1981]. According to Boehm, software cost estimation should be done through three stages: Basic COCOMO, Intermediate COCOMO, and Complete COCOMO.

Basic COCOMO Model

The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

$$\text{Effort} = a_1 \times (\text{KLOC})^{a_2} \text{ PM}$$

$$\text{Tdev} = b_1 \times (\text{Effort})^{b_2} \text{ Months}$$

Where,

- KLOC is the estimated size of the software product expressed in Kilo Lines of Code,
- a_1, a_2, b_1, b_2 are constants for each category of software products,
- Tdev is the estimated time to develop the software, expressed in months,
- Effort is the total effort required to develop the software product, expressed in person months (PMs).

The effort estimation is expressed in units of person-months (PM). It is the area under the person-month plot (as shown in fig. 11.3). It should be carefully noted that an effort of 100 PM does not imply that 100 persons should work for 1 month nor does it imply that 1 person should be employed for 100 months, but it denotes the area under the person-month curve (as shown in fig. 11.3).

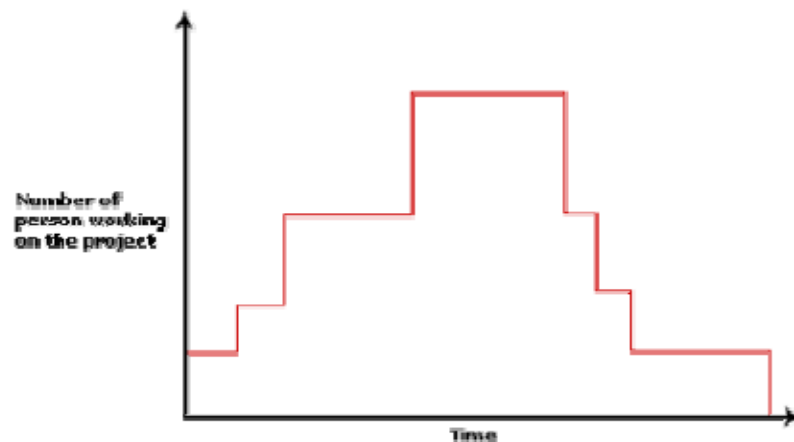


Fig. 11.3: Person-month curve

According to Boehm, every line of source text should be calculated as one LOC irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say n lines), it is considered to be nLOC. The values of a_1 , a_2 , b_1 , b_2 for different categories of products (i.e. organic, semidetached, and embedded) as given by Boehm [1981] are summarized below. He derived the above expressions by examining historical data collected from a large number of actual projects.

Estimation of development effort

For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic : **Effort = $2.4(KLOC)^{1.05}$ PM**

Semi-detached : **Effort = $3.0(KLOC)^{1.12}$ PM**

Embedded : **Effort = $3.6(KLOC)^{1.20}$ PM**

Estimation of development time

For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic : **Tdev = $2.5(Effort)^{0.38}$ Months**

Semi-detached : **Tdev = $2.5(Effort)^{0.35}$ Months**

Embedded : **Tdev = $2.5(Effort)^{0.32}$ Months**

some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes. Fig. 11.4 shows a plot of estimated effort versus product size. From fig. 11.4, we can observe that the effort is somewhat superlinear in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size.

The development time versus the product size in KLOC is plotted in fig. 11.5. From fig. 11.5, it can be observed that the development time is a sublinear function of the size of the product, i.e. when the size of the product increases by two times, the time to develop the product does not double but rises moderately. This can be explained by the fact that for larger products, a larger number of activities which can be carried out concurrently can be identified. The parallel activities can be carried out simultaneously by the engineers. This reduces the time to complete the project. Further, from fig. 11.5, it can be observed that the

development time is roughly the same for all the three categories of products. For example, a 60 KLOC program can be developed in approximately 18 months, regardless of whether it is of organic, semidetached, or embedded type.

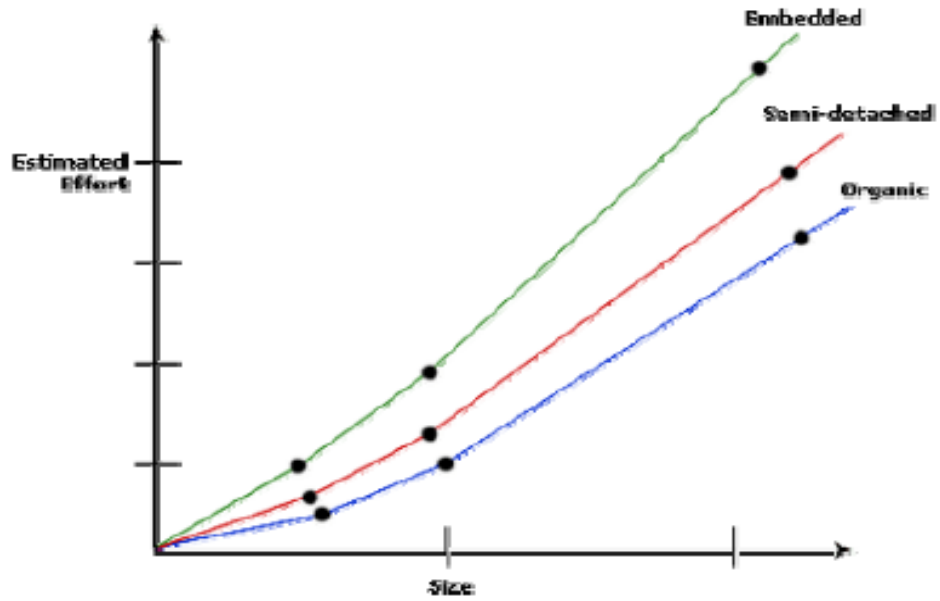


Fig. 11.4: Effort versus product size

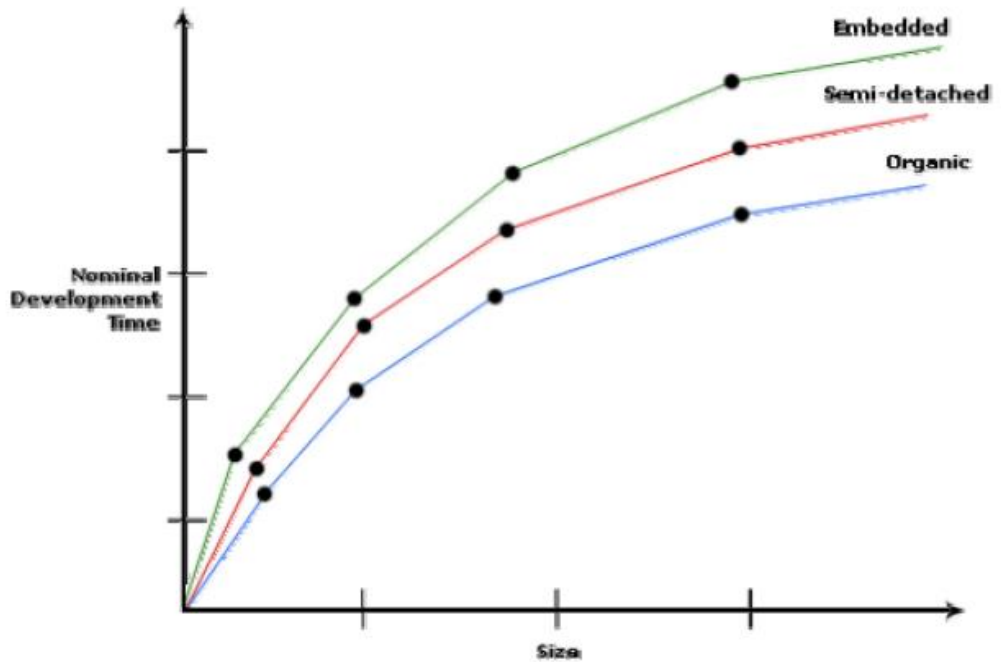


Fig. 11.5: Development time versus size

From the effort estimation, the project cost can be obtained by multiplying the required effort by the manpower cost per month. But, implicit in this project cost computation is the assumption that the entire project cost is incurred on account of the manpower cost alone. In addition to manpower cost, a project would incur costs due to hardware and software required for the project and the company overheads for administration, office space, etc.

It is important to note that the effort and the duration estimations obtained using the COCOMO model are called as nominal effort estimate and nominal duration estimate. The term nominal implies that if anyone tries to complete the project in a time shorter than the estimated duration, then the cost will increase drastically. But, if anyone completes the project over a longer period of time than the estimated, then there is almost no decrease in the estimated cost value.

Example:

Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of software engineers be Rs. 15,000/- per month. Determine the effort required to develop the software product and the nominal development time.

From the basic COCOMO estimation formula for organic software:

$$\text{Effort} = 2.4 \times (32)^{1.05} = 91 \text{ PM}$$

$$\text{Nominal development time} = 2.5 \times (91)^{0.38} = 14 \text{ months}$$

$$\text{Cost required to develop the product} = 14 \times 15,000 = \text{Rs. 210,000/-}$$

Intermediate COCOMO model

The basic COCOMO model assumes that effort and development time are functions of the product size alone. However, a host of other project parameters besides the product size affect the effort required to develop the product as well as the development time. Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account. The intermediate COCOMO model recognizes this fact and refines the initial estimate obtained using the basic COCOMO expressions by using a set of 15 cost drivers (multipliers) based on various attributes of software development. For example, if modern programming practices are used, the initial estimates are scaled downward by multiplication with a cost driver having a value less than 1. If there are stringent reliability requirements on the software product, this initial

estimate is scaled upward. Boehm requires the project manager to rate these 15 different parameters for a particular project on a scale of one to three. Then, depending on these ratings, he suggests appropriate cost driver values which should be multiplied with the initial estimate obtained using the basic COCOMO. In general, the cost drivers can be classified as being attributes of the following items:

Product: The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.

Computer: Characteristics of the computer that are considered include the execution speed required, storage space required etc.

Personnel: The attributes of development personnel that are considered include the experience level of personnel, programming capability, analysis capability, etc.

Development Environment: Development environment attributes capture the development facilities available to the developers. An important parameter that is considered is the sophistication of the automation (CASE) tools used for software development.

Complete COCOMO model

A major shortcoming of both the basic and intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up several smaller sub-systems. These sub-systems may have widely different characteristics. For example, some sub-systems may be considered as organic type, some semidetached, and some embedded. Not only that the inherent development complexity of the subsystems may be different, but also for some subsystems the reliability requirements may be high, for some the development team might have no previous experience of similar development, and so on. The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual subsystems. The cost of each subsystem is estimated separately. This approach reduces the margin of error in the final estimate.

The following development project can be considered as an example application of the complete COCOMO model. A distributed Management Information System

(MIS) product for an organization having offices at several places across the country can have the following sub-components:

- Database part
- Graphical User Interface (GUI) part
- Communication part

Of these, the communication part can be considered as embedded software. The database part could be semi-detached software, and the GUI part organic software. The costs for these three components can be estimated separately, and summed up to give the overall cost of the system.

Check Your Progress I

Mark all options which are true.

1. Normally software project planning activity is undertaken
 - a. before the development starts to plan the activities to be undertaken during development
 - b. once the development activities start
 - c. after the completion of the project monitoring and control
 - d. none of the above
2. Which of the following estimation is carried out first by a project manager during project planning?
 - a. estimation of cost
 - b. estimation of the duration of the project
 - c. project size estimation
 - d. estimation of development effort
3. Sliding Window Planning involves
 - a. planning a project before development starts
 - b. planning progressively as development proceeds
 - c. planning a project after development starts
 - d. none of the above
4. A project estimation technique based on making an educated guess of the project parameters (such as project size, effort required to develop the software, project duration, cost etc.) is
 - a. analytical estimation technique

- b. heuristic estimation technique
 - c. empirical estimation technique
 - d. none of the above
5. An example of single variable heuristic cost estimation model is
- a. Halstead's software science
 - b. basic COCOMO model
 - c. intermediate COCOMO model
 - d. complete COCOMO model

State True or False

6. Size of a project, as used in COCOMO is the size of the final executable code in bytes.
7. According to the COCOMO model, cost is the fundamental attribute of a software product, based on which size and effort are estimated.

11.3 Staffing Level Estimation and Scheduling

11.3.1 Staffing level estimation

Once the effort required to develop a software has been determined, it is necessary to determine the staffing requirement for the project. Putnam first studied the problem of what should be a proper staffing pattern for software projects. He extended the work of Norden who had earlier investigated the staffing pattern of research and development (R&D) type of projects. In order to appreciate the staffing pattern of software projects, Norden's and Putnam's results must be understood.

11.3.1.1 Norden's Work

Norden studied the staffing patterns of several R & D projects. He found that the staffing pattern can be approximated by the Rayleigh distribution curve (as shown in fig. 11.6). Norden represented the Rayleigh curve by the following equation:

$$E = K/t_d^2 * t * e^{-t^2/2 t_d^2}$$

Where E is the effort required at time t. E is an indication of the number of engineers (or the staffing level) at any particular time during the duration of the project, K is the area under the curve, and t_d is the time at which the curve attains

its maximum value. It must be remembered that the results of Norden are applicable to general R & D projects and were not meant to model the staffing pattern of software development projects.

Version 2 CSE IIT, Kharagpur

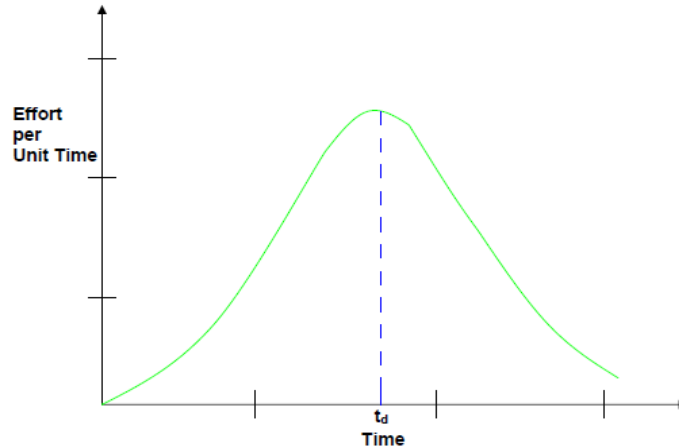


Fig. 11.6: Rayleigh curve

11.3.1.2 Putnam's Work

Putnam studied the problem of staffing of software projects and found that the software development has characteristics very similar to other R & D projects studied by Norden and that the Rayleigh-Norden curve can be used to relate the number of delivered lines of code to the effort and the time required to develop the project. By analyzing a large number of army projects, Putnam derived the following expression:

$$L = C_k K^{1/3} t_d^{4/3}$$

The various terms of this expression are as follows:

- K is the total effort expended (in PM) in the product development and L is the product size in KLOC.
- t_d corresponds to the time of system and integration testing. Therefore, t_d can be approximately considered as the time required to develop the software.
- C_k is the state of technology constant and reflects constraints that impede the progress of the programmer. Typical values of $C_k = 2$ for poor development environment (no methodology, poor documentation, and review, etc.), $C_k = 8$ for good software development environment

(software engineering principles are adhered to), $C_k = 11$ for an excellent environment (in addition to following software engineering principles, automated tools and techniques are used). The exact value of C_k for a specific project can be computed from the historical data of the organization developing it.

Putnam suggested that optimal staff build-up on a project should follow the Rayleigh curve. Only a small number of engineers are needed at the beginning of a project to carry out planning and specification tasks. As the project progresses and more detailed work is required, the number of engineers reaches a peak. After implementation and unit testing, the number of project staff falls.

However, the staff build-up should not be carried out in large installments. The team size should either be increased or decreased slowly whenever required to match the Rayleigh-Norden curve. Experience shows that a very rapid build up of project staff any time during the project development correlates with schedule slippage.

It should be clear that a constant level of manpower throughout the project duration would lead to wastage of effort and increase the time and effort required to develop the product. If a constant number of engineers are used over all the phases of a project, some phases would be overstaffed and the other phases would be understaffed causing inefficient use of manpower, leading to schedule slippage and increase in cost.

11.3.2 Effect of schedule change on cost

By analyzing a large number of army projects, Putnam derived the following expression:

$$L = C_k K^{1/3} t_d^{4/3}$$

Where, K is the total effort expended (in PM) in the product development and L is the product size in KLOC, t_d corresponds to the time of system and integration testing and C_k is the state of technology constant and reflects constraints that impede the progress of the programmer

Now by using the above expression it is obtained that,

$$K = L / C_k^3 t_d^4$$

Or,

$$K = C/t_d^4$$

For the same product size, $C = L / C_k^3$ is a constant.

$$\text{or, } K/K = t/t_d^4$$

$$\text{or, } K \propto 1/t_d^4$$

$$\text{or, cost} \propto 1/t_d$$

(as project development effort is equally proportional to project development cost)

From the above expression, it can be easily observed that when the schedule of a project is compressed, the required development effort as well as project development cost increases in proportion to the fourth power of the degree of compression. It means that a relatively small compression in delivery schedule can result in substantial penalty of human effort as well as development cost. For example, if the estimated development time is 1 year, then in order to develop the product in 6 months, the total effort required to develop the product (and hence the project cost) increases 16 times.

11.3.3 Project scheduling

Project-task scheduling is an important project planning activity. It involves deciding which tasks would be taken up when. In order to schedule the project activities, a software project manager needs to do the following:

1. Identify all the tasks needed to complete the project.
2. Break down large tasks into small activities.
3. Determine the dependency among different activities.
4. Establish the most likely estimates for the time durations necessary to complete the activities.
5. Allocate resources to activities.
6. Plan the starting and ending dates for various activities.
7. Determine the critical path. A critical path is the chain of activities that determines the duration of the project.

The first step in scheduling a software project involves identifying all the tasks necessary to complete the project. A good knowledge of the intricacies of the project and the development process helps the managers to effectively identify the important tasks of the project. Next, the large tasks are broken down into a logical set of small activities which would be assigned to different engineers. The work

breakdown structure formalism helps the manager to breakdown the tasks systematically.

After the project manager has broken down the tasks and created the work breakdown structure, he has to find the dependency among the activities. Dependency among the different activities determines the order in which the different activities would be carried out. If an activity A requires the results of another activity B, then activity A must be scheduled after activity B. In general, the task dependencies define a partial ordering among tasks, i.e. each tasks may precede a subset of other tasks, but some tasks might not have any precedence ordering defined between them (called concurrent task). The dependency among the activities are represented in the form of an activity network.

Once the activity network representation has been worked out, resources are allocated to each activity. Resource allocation is typically done using a Gantt chart. After resource allocation is done, a PERT chart representation is developed. The PERT chart representation is suitable for program monitoring and control. For task scheduling, the project manager needs to decompose the project tasks into a set of activities. The time frame when each activity is to be performed is to be determined. The end of each activity is called milestone. The project manager tracks the progress of a project by monitoring the timely completion of the milestones. If he observes that the milestones start getting delayed, then he has to carefully control the activities, so that the overall deadline can still be met.

Work breakdown structure

Work Breakdown Structure (WBS) is used to decompose a given task set recursively into small activities. WBS provides a notation for representing the major tasks need to be carried out in order to solve a problem. The root of the tree is labeled by the problem name. Each node of the tree is broken down into smaller activities that are made the children of the node. Each activity is recursively decomposed into smaller sub-activities until at the leaf level, the activities requires approximately two weeks to develop. [Fig. 11.7](#) represents the WBS of an MIS (Management Information System) software.

While breaking down a task into smaller tasks, the manager has to make some hard decisions. If a task is broken down into large number of very small activities, these can be carried out independently. Thus, it becomes possible to develop the

product faster (with the help of additional manpower). Therefore, to be able to complete a project in the least amount of time, the manager needs to break large tasks into smaller ones, expecting to find more parallelism. However, it is not useful to subdivide tasks into units which take less than a week or two to execute. Very fine subdivision means that a disproportionate amount of time must be spent on preparing and revising various charts.

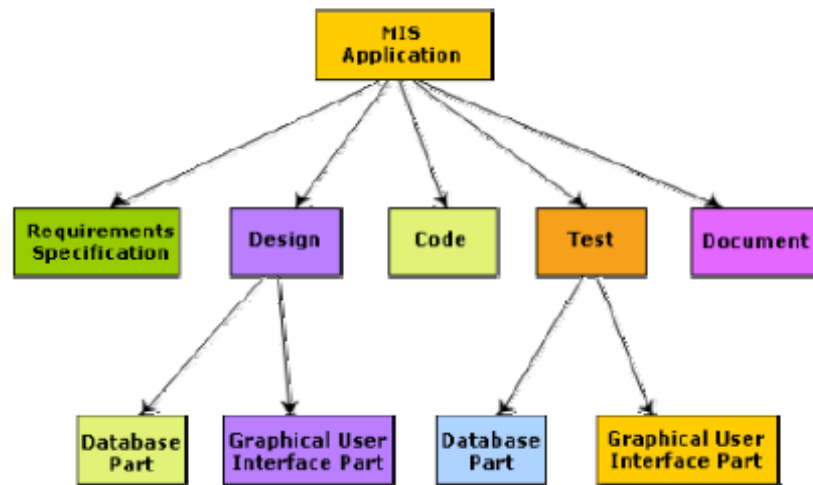


Fig. 11.7: Work breakdown structure of an MIS problem

Activity networks and critical path method

WBS representation of a project is transformed into an activity network by representing activities identified in WBS along with their interdependencies. An activity network shows the different activities making up a project, their estimated durations, and interdependencies (as shown in [fig. 11.8](#)). Each activity is represented by a rectangular node and the duration of the activity is shown alongside each task.

Managers can estimate the time durations for the different tasks in several ways. One possibility is that they can empirically assign durations to different tasks. This however is not a good idea, because software engineers often resent such unilateral decisions. A possible alternative is to let engineer himself estimate the time for an activity he can assigned to. However, some managers prefer to estimate the time for various activities themselves. Many managers believe that an aggressive schedule motivates the engineers to do a better and faster job. However, careful experiments have shown that unrealistically aggressive schedules not only cause engineers to compromise on intangible quality aspects,

but also are a cause for schedule delays. A good way to achieve accurately in estimation of the task durations without creating undue schedule pressures is to have people set their own schedules.

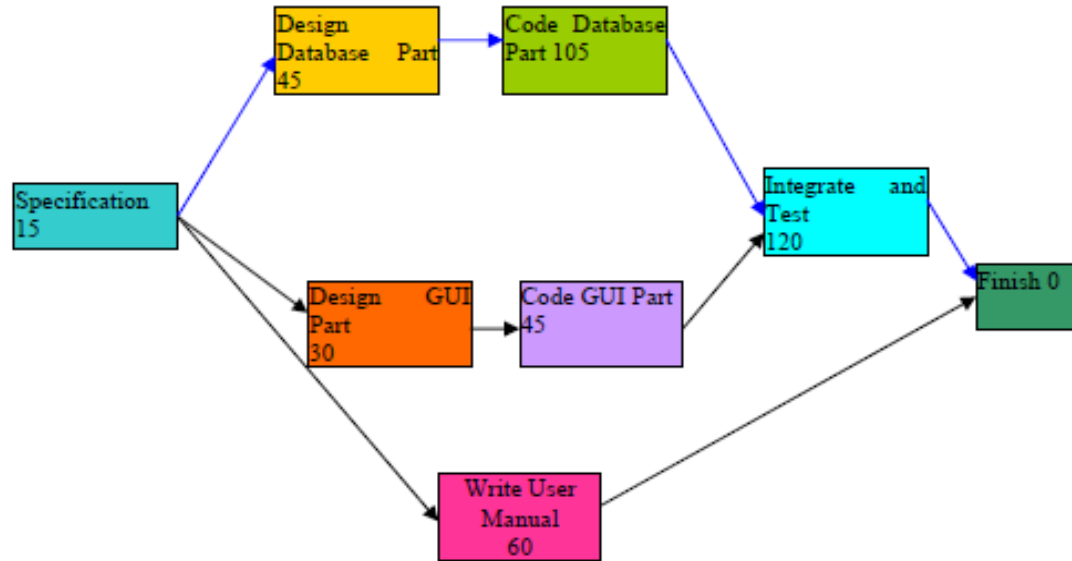


Fig. 11.8: Activity network representation of the MIS problem

Critical Path Method (CPM)

From the activity network representation following analysis can be made. The minimum time (MT) to complete the project is the maximum of all paths from start to finish. The earliest start (ES) time of a task is the maximum of all paths from the start to the task. The latest start time is the difference between MT and the maximum of all paths from this task to the finish. The earliest finish time (EF) of a task is the sum of the earliest start time of the task and the duration of the task. The latest finish (LF) time of a task can be obtained by subtracting maximum of all paths from this task to finish from MT. The slack time (ST) is $LS - EF$ and equivalently can be written as $LF - EF$. The slack time (or float time) is the total time that a task may be delayed before it will affect the end time of the project. The slack time indicates the “flexibility” in starting and completion of tasks. A critical task is one with a zero-slack time. A path from the start node to the finish node containing only critical tasks is called a critical path. These parameters for different tasks for the MIS problem are shown in the following table.

Task	ES	EF	LS	LF	ST
Specification	0	15	0	15	0
Design database	15	60	15	60	0
Design GUI part	15	45	90	120	75
Code database	60	165	60	165	0
Code GUI part	45	90	120	165	75

Gantt chart

Gantt charts are mainly used to allocate resources to activities. The resources allocated to activities include staff, hardware, and software. Gantt charts (named after its developer Henry Gantt) are useful for resource planning. A Gantt chart is a special type of bar chart where each bar represents an activity. The bars are drawn along a time line. The length of each bar is proportional to the duration of time planned for the corresponding activity.

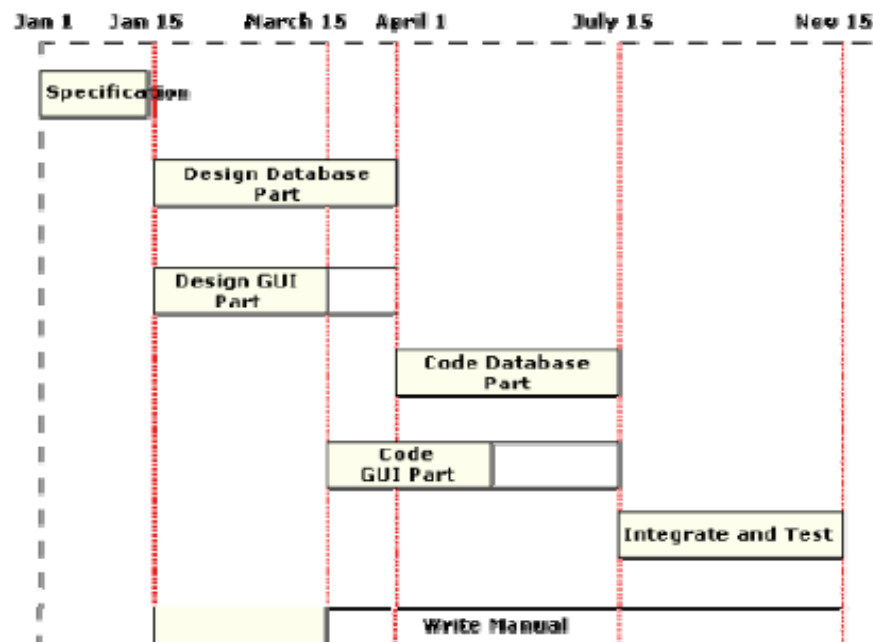


Fig. 11.9: Gantt chart representation of the MIS problem

Gantt charts are used in software project management are actually an enhanced version of the standard Gantt charts. In the Gantt charts used for software project management, each bar consists of a white part and a shaded part. The shaded part of the bar shows the length of time each task is estimated to take. The white part shows the slack time, that is, the latest time by which a task must be finished. A Gantt chart representation for the MIS problem of fig. 11.8 is shown in the fig. 11.9.

PERT chart

PERT (Project Evaluation and Review Technique) charts consist of a network of boxes and arrows. The boxes represent activities and the arrows represent task dependencies. PERT chart represents the statistical variations in the project estimates assuming a normal distribution. Thus, in a PERT chart instead of making a single estimate for each task, pessimistic, likely, and optimistic estimates are made. The boxes of PERT charts are usually annotated with the pessimistic, likely, and optimistic estimates for every task. Since all possible completion times between the minimum and maximum duration for every task has to be considered, there are not one but many critical paths, depending on the permutations of the estimates for each task. This makes critical path analysis in PERT charts very complex. A critical path in a PERT chart is shown by using thicker arrows. The PERT chart representation of the MIS problem of fig. 11.8 is shown in fig. 11.10. PERT charts are a more sophisticated form of activity chart. In activity diagrams only the estimated task durations are represented. Since, the actual durations might vary from the estimated durations, the utility of the activity diagrams are limited. Gantt chart representation of a project schedule is helpful in planning the utilization of resources, while PERT chart is useful for monitoring the timely progress of activities. Also, it is easier to identify parallel activities in a project using a PERT chart. Project managers need to identify the parallel activities in a project for assignment to different engineers.

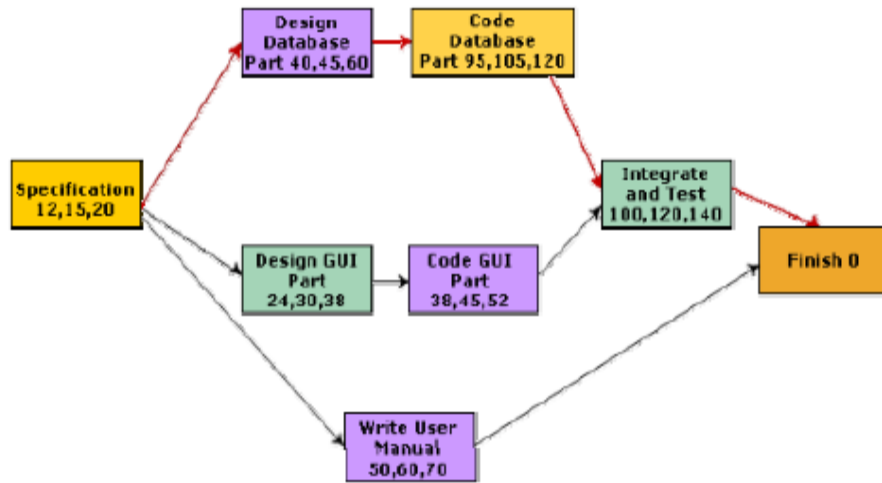


Fig. 11.10: PERT chart representation of the MIS problem

Check Your Progress II

1. Operating systems and real-time system programs can be considered as
 - a. application programs
 - b. utility programs
 - c. system programs
 - d. none of the above
2. Compilers, linkers, etc. can be considered as
 - a. application programs
 - b. utility programs
 - c. system programs
 - d. none of the above
3. Data processing programs are considered as
 - a. utility programs
 - b. system programs
 - c. application programs
 - d. none of the above
4. During project scheduling, resource allocation to different activities is done using which of the following representations?
 - a. PERT chart
 - b. activity network representation
 - c. work breakdown structure

d. Gantt chart

Mark the following as either True or False. Justify your answer.

5. If we increase the size of a software product by two times then the time required to develop that software product would be double.
6. The number of development personnel required for any software development project can be obtained by dividing the total (estimated) effort by the total (estimated) duration of the project.
7. For the development of the same product, the larger is the size of a software development team, the faster is the product development. (for simplicity, assume that all engineers are equally proficient and have exactly similar experience).
8. As a project manager it would be worthwhile on your part to reduce the project duration by half provided the customer agrees to pay for the increased manpower requirements.
9. PERT charts are a sophisticated form of activity chart.

11.4 MODEL QUESTIONS

1. List the major responsibilities of a software project manager.
2. What should be the necessary skills of a software project manager in order to perform the task of software project management?
3. When does the software planning activity start and end in software life cycle? List some important activities that a software project manager performs during software project planning.
4. What are the project related estimates performed by a project manager and also mention the order of project related estimates.
5. What do you understand by Sliding Window Planning? Explain using a few examples the types of projects for which this form of planning is especially suitable. What are its advantages over conventional planning?
6. List the important items that a Software Project Management Plan (SPMP) document should discuss.
7. Point out the major shortcomings of Lines of Code (LOC) metric in order to use it as a software project size metric.

8. List out the major shortcomings of function point metric in order to use it as a software project size metric.
9. What is the necessity of a feature point metric in the context of software project size estimation?
10. Write down the major differences in between empirical estimation technique and heuristic technique.
11. How are the software project related parameters such as program length, program vocabulary, program volume, potential minimum volume, effort to develop the project, project development time estimated using analytical estimation technique?
12. Write down the major differences between expert judgment technique and delphi cost estimation technique.
13. Write down the differences among organic, semidetached and embedded software product.
14. Differentiate among basic COCOMO model, intermediate COCOMO model and complete COCOMO model.
15. As the manager of a software project to develop a product for business application, if you estimate the effort required for completion of the project to be 50 person-months, can you complete the project by employing 50 engineers for a period of one month? Justify your answer.
16. For the same number of lines of code and the same development team size, rank the following software projects in order of their estimated development time. Show reasons behind your answer.
 - a. A text editor
 - b. An employee pay roll system
 - c. An operating system for a new computer
17. Explain Norden's model in the context of staffing requirements for a software project.
18. Explain how can Putnam's model be used to compute the change in project cost with project duration. What are the main disadvantages of using Putnam's model to compute the additional costs incurred due to schedule compression? How can you overcome them?

19. Explain why adding more manpower to an already late project makes it later.
20. Suppose you have estimated the normal development time of a moderate-sized software product to be 5 months. You have also estimated that it will cost Rs. 50,000/- to develop the software product. Now, the customer comes and tells you that he wants you to accelerate the delivery time by 10%. How much additional cost would you charge the customer for this accelerated delivery? Irrespective of whether you take less time or more time to develop the product, you are essentially developing the same product. Why then does the effort depend on the duration over which you develop the product?
21. How does the change of project duration affect the overall project development effort and development cost?
22. Write down the necessary tasks performed by a project manager in order to perform project scheduling.
23. Write down the major differences between work breakdown structure and activity network model.
24. Explain critical path method.
25. Explain when you should use PERT charts and when you should use Gantt charts while you are performing the duties of a project manager.

UNIT XII: SOFTWARE PROJECT MONITORING AND CONTROL

12.0 LEARNING OBJECTIVES

At the end of this lesson the student would be able to:

- Explain the necessity of a suitable organization structure.
- Differentiate between functional format and project format in the context of organization structure.
- Identify the advantages of a functional organization over a project organization.
- Explain why the functional format is not suitable for small organizations handling just one or two projects.
- Identify the important types of team structures of an organization.
- Explain what is meant by egoless programming technique.
- Identify the characteristics of a good software engineer.
- Identify the main categories of risks which can affect a software project.
- Explain how to assess a project risk.
- Identify the main strategies to plan for risk containment.
- Explain what risk leverage is.
- Explain how to handle the risk of schedule slippage.
- Explain what is meant by configuration of a software product.
- Differentiate among release, version and revision of a software product.
- Explain the necessity of software configuration management.
- Identify the principal activities of software configuration management.
- Identify the activities carried out during configuration identification.
- Identify the activities carried out during configuration control.
- Identify the popular configuration management tools.

12.1. ORGANIZATION AND TEAM STRUCTURES

12.1.1 Organization structure

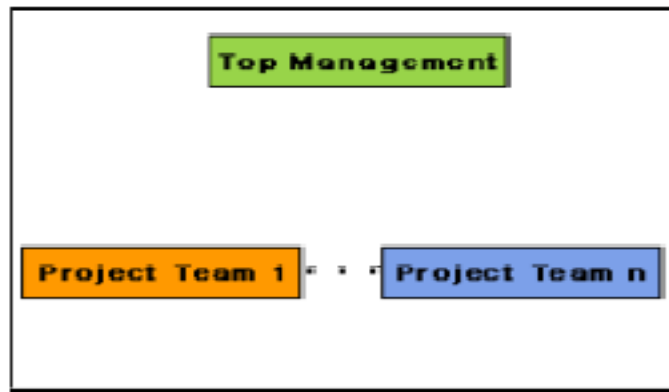
Usually every software development organization handles several projects at any time. Software organizations assign different teams of engineers to handle different software projects. Each type of organization structure has its own advantages and disadvantages so the issue “how is the organization as a whole structured?” must be taken into consideration so that each software project can be finished before its deadline.

12.1.1.1 Functional format vs. project format

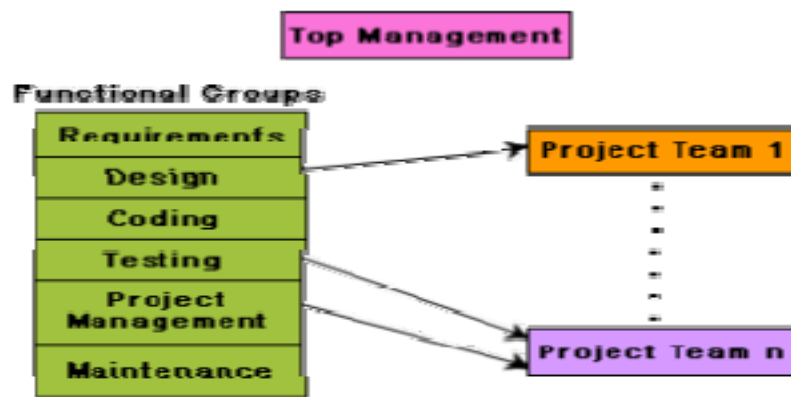
There are essentially two broad ways in which a software development organization can be structured: functional format and project format. In the project format, the project development staff are divided based on the project for which they work (as shown in fig. 12.1). In the functional format, the development staff are divided based on the functional group to which they belong. The different projects borrow engineers from the required functional groups for specific phases to be undertaken in the project and return them to the functional group upon the completion of the phase.

In the functional format, different teams of programmers perform different phases of a project. For example, one team might do the requirements specification, another do the design, and so on. The partially completed product passes from one team to another as the project evolves. Therefore, the functional format requires considerable communication among the different teams because the work of one team must be clearly understood by the subsequent teams working on the project. This requires good quality documentation to be produced after every activity.

In the project format, a set of engineers is assigned to the project at the start of the project and they remain with the project till the completion of the project. Thus, the same team carries out all the life cycle activities. Obviously, the functional format requires more communication among teams than the project format, because one team must understand the work done by the previous teams.



(a) Project Organization



(b) Functional Organization

Fig. 12.1: Schematic representation of the functional and project organization

12.1.1.2 Advantages of functional organization over project organization

Even though greater communication among the team members may appear as an avoidable overhead, the functional format has many advantages. The main advantages of a functional organization are:

- Ease of staffing
- Production of good quality documents
- Job specialization
- Efficient handling of the problems associated with manpower turnover.

The functional organization allows the engineers to become specialists in particular roles, e.g. requirements analysis, design, coding, testing, maintenance, etc. They perform these roles again and again for different projects and develop

deep insights to their work. It also results in more attention being paid to proper documentation at the end of a phase because of the greater need for clear communication as between teams doing different phases. The functional organization also provides an efficient solution to the staffing problem. We have already seen that the staffing pattern should approximately follow the Rayleigh distribution for efficient utilization of the personnel by minimizing their wait times. The project staffing problem is eased significantly because personnel can be brought onto a project as needed, and returned to the functional group when they are no more needed. This possibly is the most important advantage of the functional organization. A project organization structure forces the manager to take in almost a constant number of engineers for the entire duration of his project. This results in engineers idling in the initial phase of the software development and are under tremendous pressure in the later phase of the development. A further advantage of the functional organization is that it is more effective in handling the problem of manpower turnover. This is because engineers can be brought in from the functional pool when needed. Also, this organization mandates production of good quality documents, so new engineers can quickly get used to the work already done.

12.1.1.3 Unsuitability of functional format in small organizations

In spite of several advantages of the functional organization, it is not very popular in the software industry. The apparent paradox is not difficult to explain. The project format provides job rotation to the team members. That is, each team member takes on the role of the designer, coder, tester, etc during the course of the project. On the other hand, considering the present skill shortage, it would be very difficult for the functional organizations to fill in slots for some roles such as maintenance, testing, and coding groups. Also, another problem with the functional organization is that if an organization handles projects requiring knowledge of specialized domain areas, then these domain experts cannot be brought in and out of the project for the different phases, unless the company handles a large number of such projects. Also, for obvious reasons the functional format is not suitable for small organizations handling just one or two projects.

12.1.2 Team structures

Team structure addresses the issue of organization of the individual project teams. There are some possible ways in which the individual project teams can be organized. There are mainly three formal team structures: chief programmer, democratic, and the mixed team organizations although several other variations to these structures are possible. Problems of different complexities and sizes often require different team structures for chief solution.

12.1.2.1 Chief Programmer Team

In this team organization, a senior engineer provides the technical leadership and is designated as the chief programmer. The chief programmer partitions the task into small activities and assigns them to the team members. He also verifies and integrates the products developed by different team members. The structure of the chief programmer team is shown in fig. 12.2. The chief programmer provides an authority, and this structure is arguably more efficient than the democratic team for well-understood problems. However, the chief programmer team leads to lower team morale, since team-members work under the constant supervision of the chief programmer. This also inhibits their original thinking. The chief programmer team is subject to single point failure since too much responsibility and authority is assigned to the chief programmer.

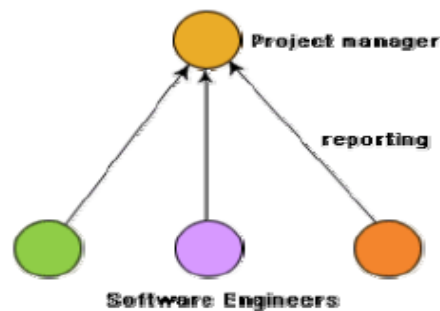


Fig. 12.2: Chief programmer team structure

The chief programmer team is probably the most efficient way of completing simple and small projects since the chief programmer can work out a satisfactory design and ask the programmers to code different modules of his design solution. For example, suppose an organization has successfully completed many simple MIS projects. Then, for a similar MIS project, chief programmer team structure can be adopted. The chief programmer team structure works well when the task is

within the intellectual grasp of a single individual. However, even for simple and well-understood problems, an organization must be selective in adopting the chief programmer structure. The chief programmer team structure should not be used unless the importance of early project completion outweighs other factors such as team morale, personal developments, life-cycle cost etc.

12.1.2.2 Democratic Team

The democratic team structure, as the name implies, does not enforce any formal team hierarchy (as shown in fig. 12.3). Typically, a manager provides the administrative leadership. At different times, different members of the group provide technical leadership.

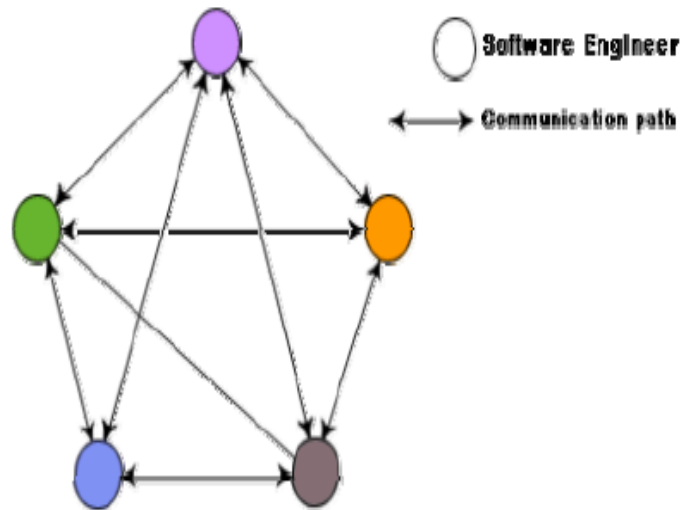


Fig. 12.3: Democratic team structure

The democratic organization leads to higher morale and job satisfaction. Consequently, it suffers from less man-power turnover. Also, democratic team structure is appropriate for less understood problems, since a group of engineers can invent better solutions than a single individual as in a chief programmer team. A democratic team structure is suitable for projects requiring less than five or six engineers and for research-oriented projects. For large sized projects, a pure democratic organization tends to become chaotic. The democratic team organization encourages egoless programming as programmers can share and review one another's work.

12.1.2.3 Mixed Control Team Organization

The mixed team organization, as the name implies, draws upon the ideas from both the democratic organization and the chief-programmer organization. The mixed control team organization is shown pictorially in fig. 12.4. This team organization incorporates both hierarchical reporting and democratic set up. In fig. 12.4, the democratic connections are shown as dashed lines and the reporting structure is shown using solid arrows. The mixed control team organization is suitable for large team sizes. The democratic arrangement at the senior engineers level is used to decompose the problem into small parts. Each democratic setup at the programmer level attempts solution to a single part. Thus, this team organization is eminently suited to handle large and complex programs. This team structure is extremely popular and is being used in many software development companies.

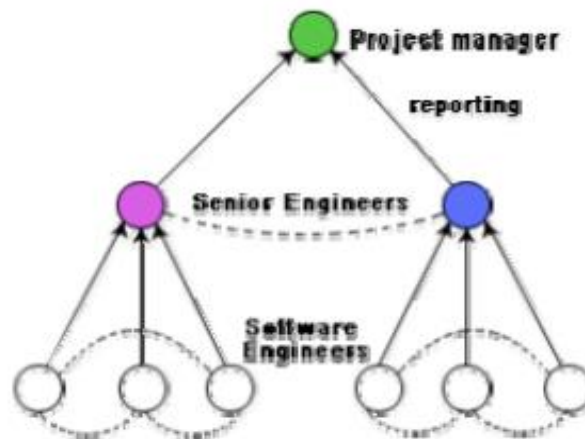


Fig. 12.4: Mixed team structure

12.1.2.4 Egoless programming technique

Ordinarily, the human psychology makes an individual take pride in everything he creates using original thinking. Software development requires original thinking too, although of a different type. The human psychology makes one emotionally involved with his creation and hinders him from objective examination of his creations. Just like temperamental artists, programmers find it extremely difficult to locate bugs in their own programs or flaws in their own design. Therefore, the best way to find problems in a design or code is to have someone review it. Often, having to explain one's program to someone else gives a person enough

objectivity to find out what might have gone wrong. This observation is the basic idea behind code walk throughs. An application of this, is to encourage a democratic team to think that the design, code, and other deliverables to belong to the entire group. This is called egoless programming technique.

12.1.3 Characteristics of a good software engineer

The attributes that good software engineers should possess are as follows:

- Exposure to systematic techniques, i.e. familiarity with software engineering principles.
- Good technical knowledge of the project areas (Domain knowledge).
- Good programming abilities.
- Good communication skills. These skills comprise of oral, written, and interpersonal skills.
- High motivation.
- Sound knowledge of fundamentals of computer science.
- Intelligence.
- Ability to work in a team.
- Discipline, etc.

Studies show that these attributes vary as much as 1:30 for poor and bright candidates. An experiment conducted by Sackman [1968] shows that the ratio of coding hour for the worst to the best programmers is 25:1, and the ratio of debugging hours is 28:1. Also, the ability of a software engineer to arrive at the design of the software from a problem description varies greatly with respect to the parameters of quality and time.

Technical knowledge in the area of the project (domain knowledge) is an important factor determining the productivity of an individual for a particular project, and the quality of the product that he develops. A programmer having a thorough knowledge of database application (e.g. MIS) may turn out to be a poor data communication engineer. Lack of familiarity with the application areas can result in low productivity and poor quality of the product.

Since software development is a group activity, it is vital for a software engineer to possess three main kinds of communication skills: Oral, Written, and Interpersonal. A software engineer not only needs to effectively communicate with

his teammates (e.g. reviews, walk throughs, and other team communications) but may also have to communicate with the customer to gather product requirements. Poor interpersonal skills hamper these vital activities and often show up as poor quality of the product and low productivity. Software engineers are also required at times to make presentations to the managers and to the customers. This requires a different kind of communication skill (oral communication skill). A software engineer is also expected to document his work (design, code, test, etc.) as well as write the users' manual, training manual, installation manual, maintenance manual, etc. This requires good written communication skill.

Motivation level of software engineers is another crucial factor contributing to his work quality and productivity. Even though no systematic studies have been reported in this regard, it is generally agreed that even bright engineers may turn out to be poor performers when they have lack motivation. An average engineer who can work with a single mind track can outperform other engineers, higher incentives and better working conditions have only limited affect on their motivation levels. Motivation is to a great extent determined by personal traits, family and social backgrounds, etc.

Check Your Progress I

Mark all options which are true.

1. When are the software project monitoring and control activities undertaken?
 - a. before the development starts to plan the activities to be undertaken during development
 - b. once the development activities start with the aim of ensuring that the development proceeds as per plan
 - c. at the end of the development
 - d. none of the above
2. Job specialization is one of the main advantages in case of which organization structure?
 - a. project format
 - b. function format
 - c. either project format or function format

- d. both of project format and function format
- 3. Pure egoless programming is encouraged by which team organization?
 - a. chief programmer team structure
 - b. democratic team structure
 - c. mixed control team structure
 - d. none of the above
- 4. In which type of team organization a single point failure of development occurs when an individual leaves the team?
 - a. chief programmer team structure
 - b. democratic team structure
 - c. mixed control team structure
 - d. none of the above

Mark the following as either True or False. Justify your answer.

- 5. Software organizations achieve efficient manpower utilization by adopting a project-based organization structure.
- 6. In order to handle complex software projects requiring knowledge of specialized domain areas, software organization would surely prefer functional organization structure.
- 7. The pure democratic team organization is well suited to handle extremely large and complex projects.
- 8. Technical knowledge in the area of the project i.e. domain knowledge is an important factor determining the productivity of an individual for a particular project.
- 9. For high productivity of a developer, strong interpersonal skills are essential.

12.2. RISK MANAGEMENT AND SOFTWARE CONFIGURATION MANAGEMENT

12.2.1 Risk management

A software project can be affected by a large variety of risks. In order to be able to systematically identify the important risks which might affect a software project, it is necessary to categorize risks into different classes. The project manager can then examine which risks from each class are relevant to the project. There are three main categories of risks which can affect a software project:

Project risks. Project risks concern various forms of budgetary, schedule, personnel, resource, and customer-related problems. An important project risk is schedule slippage. Since, software is intangible, it is very difficult to monitor and control a software project. It is very difficult to control something which cannot be seen. For any manufacturing project, such as manufacturing of cars, the project manager can see the product taking shape. He can for instance, see that the engine is fitted, after that the doors are fitted, the car is getting painted, etc. Thus, he can easily assess the progress of the work and control it. The invisibility of the product being developed is an important reason why many software projects suffer from the risk of schedule slippage.

Technical risks. Technical risks concern potential design, implementation, interfacing, testing, and maintenance problems. Technical risks also include ambiguous specification, incomplete specification, changing specification, technical uncertainty, and technical obsolescence. Most technical risks occur due to the development team's insufficient knowledge about the project.

Business risks. This type of risks includes risks of building an excellent product that no one wants, losing budgetary or personnel commitments, etc.

12.2.1.1 Risk Assessment

The objective of risk assessment is to rank the risks in terms of their damage causing potential. For risk assessment, first each risk should be rated in two ways:

- The likelihood of a risk coming true (denoted as r).
- The consequence of the problems associated with that risk (denoted as s).

Based on these two factors, the priority of each risk can be computed:

$$p = r * s$$

Where, p is the priority with which the risk must be handled, r is the probability of the risk becoming true, and s is the severity of damage caused due to the risk

becoming true. If all identified risks are prioritized, then the most likely and damaging risks can be handled first and more comprehensive risk abatement procedures can be designed for these risks.

12.2.1.2 Risk containment

After all the identified risks of a project are assessed, plans must be made to contain the most damaging and the most likely risks. Different risks require different containment procedures. In fact, most risks require ingenuity on the part of the project manager in tackling the risk. There are three main strategies to plan for risk containment:

Avoid the risk: This may take several forms such as discussing with the customer to change the requirements to reduce the scope of the work, giving incentives to the engineers to avoid the risk of manpower turnover, etc.

Transfer the risk: This strategy involves getting the risky component developed by a third party, buying insurance cover, etc.

Risk reduction: This involves planning ways to contain the damage due to a risk. For example, if there is risk that some key personnel might leave, new recruitment may be planned.

12.2.1.3 Risk leverage

To choose between the different strategies of handling a risk, the project manager must consider the cost of handling the risk and the corresponding reduction of risk. For this the risk leverage of the different risks can be computed. Risk leverage is the difference in risk exposure divided by the cost of reducing the risk. More formally,

risk leverage = (risk exposure before reduction – risk exposure after reduction) / (cost of reduction)

12.2.1.4 Risk related to schedule slippage

Even though there are three broad ways to handle any risk, but still risk handling requires a lot of ingenuity on the part of a project manager. As an example, it can be considered the options available to contain an important type of risk that occurs in many software projects – that of schedule slippage. Risks relating to schedule

slippage arise primarily due to the intangible nature of software. Therefore, these can be dealt with by increasing the visibility of the software product. Visibility of a software product can be increased by producing relevant documents during the development process wherever meaningful and getting these documents reviewed by an appropriate team. Milestones should be placed at regular intervals through a software engineering process to provide a manager with regular indication of progress. Completion of a phase of the development process before followed need not be the only milestones. Every phase can be broken down to reasonable-sized tasks and milestones can be scheduled for these tasks too. A milestone is reached, once documentation produced as part of a software engineering task is produced and gets successfully reviewed. Milestones need not be placed for every activity. An approximate rule of thumb is to set a milestone every 10 to 15 days.

12.2.2 Software configuration management

The results (also called as the deliverables) of a large software development effort typically consist of a large number of objects, e.g. source code, design document, SRS document, test document, user's manual, etc. These objects are usually referred to and modified by a number of software engineers through out the life cycle of the software. The state of all these objects at any point of time is called the configuration of the software product. The state of each deliverable object changes as development progresses and also as bugs are detected and fixed.

12.2.2.1 Release vs. Version vs. Revision

A new version of a software is created when there is a significant change in functionality, technology, or the hardware it runs on, etc. On the other hand a new revision of a software refers to minor bug fix in that software. A new release is created if there is only a bug fix, minor enhancements to the functionality, usability, etc.

For example, one version of a mathematical computation package might run on Unix-based machines, another on Microsoft Windows and so on. As a software is released and used by the customer, errors are discovered that need correction. Enhancements to the functionalities of the software may also be needed. A new release of software is an improved system intended to replace an old one. Often

systems are described as version m, release n; or simple m.n. Formally, a history relation is version of can be defined between objects. This relation can be split into two sub relations *is revision of* and *is variant of*.

12.2.2.2 Necessity of software configuration management

There are several reasons for putting an object under configuration management. But, possibly the most important reason for configuration management is to control the access to the different deliverable objects. Unless strict discipline is enforced regarding updation and storage of different objects, several problems appear. The following are some of the important problems that appear if configuration management is not used.

Inconsistency problem when the objects are replicated. A scenario can be considered where every software engineer has a personal copy of an object (e.g. source code). As each engineer makes changes to his local copy, he is expected to intimate them to other engineers, so that the changes in interfaces are uniformly changed across all modules. However, many times an engineer makes changes to the interfaces in his local copies and forgets to intimate other teammates about the changes. This makes the different copies of the object inconsistent. Finally, when the product is integrated, it does not work. Therefore, when several team members work on developing an object, it is necessary for them to work on a single copy of the object, otherwise inconsistency may arise.

Problems associated with concurrent access. Suppose there is a single copy of a problem module, and several engineers are working on it. Two engineers may simultaneously carry out changes to different portions of the same module, and while saving overwrite each other. Though the problem associated with concurrent access to program code has been explained, similar problems occur for any other deliverable object.

Providing a stable development environment. When a project is underway, the team members need a stable environment to make progress. Suppose somebody is trying to integrate module A, with the modules B and C, he cannot make progress if developer of module C keeps changing C; this can be especially frustrating if a change to module C forces him to recompile A. When an effective configuration management is in place, the manager freezes the objects to form a base line. When

anyone needs any of the objects under configuration control, he is provided with a copy of the base line item. The requester makes changes to his private copy. Only after the requester is through with all modifications to his private copy, the configuration is updated and a new base line gets formed instantly. This establishes a baseline for others to use and depend on. Also, configuration may be frozen periodically. Freezing a configuration may involve archiving everything needed to rebuild it. (Archiving means copying to a safe place such as a magnetic tape).

System accounting and maintaining status information. System accounting keeps track of who made a particular change and when the change was made.

Handling variants. Existence of variants of a software product causes some peculiar problems. Suppose somebody has several variants of the same module, and finds a bug in one of them. Then, it has to be fixed in all versions and revisions. To do it efficiently, he should not have to fix it in each and every version and revision of the software separately.

12.2.2.3 Software configuration management activities

Normally, a project manager performs the configuration management activity by using an automated configuration management tool. A configuration management tool provides automated support for overcoming all the problems mentioned above. In addition, a configuration management tool helps to keep track of various deliverable objects, so that the project manager can quickly and unambiguously determine the current state of the project. The configuration management tool enables the engineers to change the various components in a controlled manner.

Configuration management is carried out through two principal activities:

- Configuration identification involves deciding which parts of the system should be kept track of.
- Configuration control ensures that changes to a system happen smoothly.

Configuration identification

The project manager normally classifies the objects associated with a software development effort into three main categories: controlled, precontrolled, and uncontrolled. Controlled objects are those which are already put under

configuration control. One must follow some formal procedures to change them. Precontrolled objects are not yet under configuration control, but will eventually be under configuration control. Uncontrolled objects are not and will not be subjected to configuration control. Controllable objects include both controlled and precontrolled objects. Typical controllable objects include:

- Requirements specification document
- Design documents
- Tools used to build the system, such as compilers, linkers, lexical analyzers, parsers, etc.
- Source code for each module
- Test cases
- Problem reports

The configuration management plan is written during the project planning phase and it lists all controlled objects. The managers who develop the plan must strike a balance between controlling too much, and controlling too little. If too much is controlled, overheads due to configuration management increase to unreasonably high levels. On the other hand, controlling too little might lead to confusion when something changes.

Configuration control

Configuration control is the process of managing changes to controlled objects. Configuration control is the part of a configuration management system that most directly affects the day-to-day operations of developers. The configuration control system prevents unauthorized changes to any controlled objects. In order to change a controlled object such as a module, a developer can get a private copy of the module by a reserve operation as shown in fig. 12.5. Configuration management tools allow only one person to reserve a module at a time. Once an object is reserved, it does not allow any one else to reserve this module until the reserved module is restored as shown in fig. 12.5. Thus, by preventing more than one engineer to simultaneously reserve a module, the problems associated with concurrent access are solved.

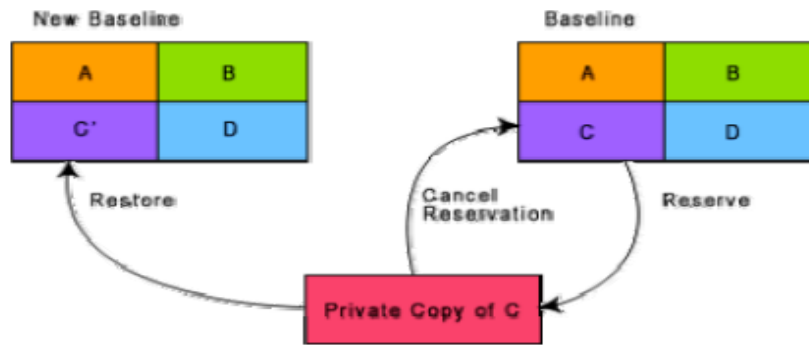


Fig. 12.5: Reserve and restore operation in configuration control

It can be shown how the changes to any object that is under configuration control can be achieved. The engineer needing to change a module first obtains a private copy of the module through a reserve operation. Then, he carries out all necessary changes on this private copy. However, restoring the changed module to the system configuration requires the permission of a change control board (CCB). The CCB is usually constituted from among the development team members. For every change that needs to be carried out, the CCB reviews the changes made to the controlled object and certifies several things about the change:

1. Change is well-motivated.
2. Developer has considered and documented the effects of the change.
3. Changes interact well with the changes made by other developers.
4. Appropriate people (CCB) have validated the change, e.g. someone has tested the changed code, and has verified that the change is consistent with the requirement.

The change control board (CCB) sounds like a group of people. However, except for very large projects, the functions of the change control board are normally discharged by the project manager himself or some senior member of the development team. Once the CCB reviews the changes to the module, the project manager updates the old base line through a restore operation (as shown in fig. 12.5). A configuration control tool does not allow a developer to replace an object he has reserved with his local copy unless he gets an authorization from the CCB. By constraining the developers' ability to replace reserved objects, a stable environment is achieved. Since a configuration management tool allows only one engineer to work on one module at any one time, problem of accidental

overwriting is eliminated. Also, since only the manager can update the baseline after the CCB approval, unintentional changes are eliminated.

12.2.2.4 Configuration management tools

SCCS and RCS are two popular configuration management tools available on most UNIX systems. SCCS or RCS can be used for controlling and managing different versions of text files. SCCS and RCS do not handle binary files (i.e. executable files, documents, files containing diagrams, etc.) SCCS and RCS provide an efficient way of storing versions that minimizes the amount of occupied disk space. Suppose, a module MOD is present in three versions MOD1.1, MOD1.2, and MOD1.3. Then, SCCS and RCS stores the original module MOD1.1 together with changes needed to transform MOD1.1 into MOD1.2 and MOD1.2 to MOD1.3. The changes needed to transform each base lined file to the next version are stored and are called deltas. The main reason behind storing the deltas rather than storing the full version files is to save disk space.

The change control facilities provided by SCCS and RCS include the ability to incorporate restrictions on the set of individuals who can create new versions, and facilities for checking components in and out (i.e. reserve and restore operations). Individual developers check out components and modify them. After they have made all necessary changes to a module and after the changes have been reviewed, they check in the changed module into SCCS or RCS. Revisions are denoted by numbers in ascending order, e.g., 1.1, 1.2, 1.3 etc. It is also possible to create variants or revisions of a component by creating a fork in the development history.

Check Your Progress II

Mark all options which are true.

1. The primary purpose of risk management is
 - a. risk containment
 - b. risk assessment
 - c. risk identification
 - d. all of the above
2. Schedule slippage is a type of

- a. business risk
 - b. project risk
 - c. technical risk
 - d. none of the above
3. A development team's insufficient knowledge of the product being developed is one of the main factors contributing to
- a. business risk
 - b. project risk
 - c. technical risk
 - d. none of the above
4. Visibility of a software product can be increased by
- a. producing relevant documents during the development process
 - b. properly reviewing those relevant documents by an expert team
 - c. placing milestones at regular intervals through a software engineering process
 - d. all of the above
5. A new version of a software is produced when there is a
- a. minor bug fix
 - b. minor enhancements to the functionality, usability etc.
 - c. significant change in functionality, technology, or the hardware the software runs on
 - d. all of the above
6. A revision of a software refers to
- a. minor bug fix
 - b. minor enhancements to the functionality, usability etc.
 - c. significant change in functionality, technology, or the hardware the software runs on
 - d. all of the above
7. If configuration management is not during a software development effort used then which of the following problems are likely to appear:
- a. inconsistency problem when the objects are replicated
 - b. problems associated with concurrent access
 - c. problems with handling several variants

- d. all of the above
8. If we develop several versions of the same software product without using any configuration management tools then the problems that we would face are
- a. bug fixing in any version would require manually fixing the same bug in all versions
 - b. large storage requirements would be needed
 - c. difficulty in keeping track the updated configurations of various versions
 - d. all of the above
9. Revisions to different software products are handled by
- a. version control
 - b. change control
 - c. neither version control nor change control
 - d. both version control and change control
10. For effective configuration control, in order to change a controlled object such as a module, a developer can get a private copy of the module by using:
- a. restore operation
 - b. reserve operation
 - c. update operation
 - d. none of the above

Mark the following as either True or False. Justify your answer.

11. It is possible to do the configuration management of a software project without using an automated tool.
12. A version and a release of a software product are synonyms.
13. Source Code Control System (SCCS) and RCS provide an efficient way of storing versions that minimizes the amount of occupied disk space.

12.3 MODEL QUESTIONS

The following questions have been designed to test the objectives identified for this module:

1. Compare the relative advantages of the functional and the project approaches to organizing a development center.
2. Suppose you are the chief executive officer (CEO) of a software development center. Which organization structure would you select for your organization and why?
3. Why the functional format is not suitable for small organizations handling just one or two projects?
4. Name the different ways in which software development teams are organized. For the development of a challenging satellite-based mobile communication product which type of project team organization would you recommend? Justify your answer.
5. Suppose you are the project manager of a large software product development team and you have to make a choice between democratic and chief programmer team organizations, which one would you adopt for your team? Explain the reasoning behind your answer.
6. What is egoless programming? How can it be realized?
7. In what units can you measure the productivity of a software development team? List three important factors that affect the productivity of a software development team.
8. As a project manager, identify the characteristics that you would look for in a software engineer while trying to select personnel for your team.
9. List three common types of risks that a typical software project might suffer from.
10. Explain how you can identify the risks that your project is susceptible to. Suppose you are the project manager of a large software development project, point out the main steps you would follow to manage risks in your software project.
11. What is meant by risk leverage?
12. Schedule slippage is a very common form of risk that almost every project manager has to overcome. Explain how would you manage the risk of schedule slippage as the project manager of a medium-sized project?
13. What do you understand by software configuration?
14. Differentiate among release, version and revision of a software product.

15. Why is software configuration management crucial to the success of large software product development projects?
16. How can you effectively manage software configuration?
17. Discuss how SCCS and RCS can be used to efficiently manage the configuration of source code.

UNIT XIII: SOFTWARE RELIABILITY AND QUALITY MANAGEMENT

Specific Instructional Objectives

At the end of this lesson the student would be able to:

- Differentiate between a repeatable software development organization and a non-repeatable software development organization.
- What is the relationship between the number of latent errors in a software system and its reliability?
- Identify the main reasons for why software reliability is difficult to measure.
- Explain how the characteristics of hardware reliability and software reliability differ.
- Identify the reliability metrics which can be used to quantify the reliability of software products.
- Identify the different types of failures of software products.
- Explain the reliability growth models of a software product.
- Identify the primary objective of statistical testing.
- Define what is meant by the operation profile of a software product.
- Identify the steps in which statistical testing is performed on a software product.
- Identify the advantages and disadvantages of statistical testing.
- Identify the main quality factors of a software product.
- Explain what is meant by quality management system.
- Identify the phases over which quality management system has evolved in the last century.
- State what is meant by ISO 9000 certification.
- Identify the different industries to which the different types of ISO 9000 quality standards can be applied.

- Differentiate between the characteristics of software products and other type of products that make managing a software development effort difficult.
- Identify the reasons why obtaining ISO 9000 certification is beneficial to a software development organization.
- Explain the main requirements that a software development organization must satisfy for getting ISO 9001 certification.
- Identify the salient features of ISO 9001 certification.
- Identify the shortcomings of ISO 9000 certification.
- Identify the different levels of SEI Capability Maturity Model.
- Explain the key process areas of a software organization provided by SEI CMM model.
- Differentiate between ISO 9000 certification and SEI CMM.
- Explain the type of systems to which the SEI CMM model of quality management is applicable.
- Explain what personal software process is.
- Explain what six sigma is.

13.1 SOFTWARE RELIABILITY ISSUES

13.1.1 Repeatable vs. non-repeatable software development organization

A repeatable software development organization is one in which the software development process is person-independent. In a non-repeatable software development organization, a software development project becomes successful primarily due to the initiative, effort, brilliance, or enthusiasm displayed by certain individuals. Thus, in a non-repeatable software development organization, the chances of successful completion of a software project is to a great extent depends on the team members.

13.1.2 Software reliability

Reliability of a software product essentially denotes its trustworthiness or dependability. Alternatively, reliability of a software product can also be defined as the probability of the product working “correctly” over a given period of time.

It is obvious that a software product having a large number of defects is unreliable. It is also clear that the reliability of a system improves, if the number of defects in it is reduced. However, there is no simple relationship between the observed system reliability and the number of latent defects in the system. For example, removing errors from parts of a software which are rarely executed makes little difference to the perceived reliability of the product. It has been experimentally observed by analyzing the behavior of a large number of programs that 90% of the execution time of a typical program is spent in executing only 10% of the instructions in the program. These most used 10% instructions are often called the core of the program. The rest 90% of the program statements are called non-core and are executed only for 10% of the total execution time. It therefore may not be very surprising to note that removing 60% product defects from the least used parts of a system would typically lead to only 3% improvement to the product reliability. It is clear that the quantity by which the overall reliability of a program improves due to the correction of a single error depends on how frequently the corresponding instruction is executed.

Thus, reliability of a product depends not only on the number of latent errors but also on the exact location of the errors. Apart from this, reliability also depends upon how the product is used, i.e. on its execution profile. If it is selected input data to the system such that only the “correctly” implemented functions are executed, none of the errors will be exposed and the perceived reliability of the product will be high. On the other hand, if the input data is selected such that only those functions which contain errors are invoked, the perceived reliability of the system will be very low.

13.1.2.1 Reasons for software reliability being difficult to measure

The reasons why software reliability is difficult to measure can be summarized as follows:

- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
- The perceived reliability of a software product is highly observer-dependent.
- The reliability of a product keeps changing as errors are detected and fixed.

13.1.2.2 Hardware reliability vs. software reliability differ

Reliability behavior for hardware and software are very different. For example, hardware failures are inherently different from software failures. Most hardware failures are due to component wear and tear. A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix hardware faults, one has to either replace or repair the failed part. On the other hand, a software product would continue to fail until the error is tracked down and either the design or the code is changed. For this reason, when a hardware is repaired its reliability is maintained at the level that existed before the failure occurred; whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug introduces new errors). To put this fact in a different perspective, hardware reliability study is concerned with stability (for example, inter-failure times remain constant). On the other hand, software reliability study aims at reliability growth (i.e. inter-failure times increase). The change of failure rate over the product lifetime for a typical hardware and a software product are sketched in fig. 13.1. For hardware products, it can be observed that failure rate is high initially but decreases as the faulty components are identified and removed. The system then enters its useful life. After some time (called product life time) the components wear out, and the failure rate increases. This gives the plot of hardware reliability over time its characteristics “bath tub” shape. On the other hand, for software the failure rate is at it’s highest during integration and test. As the system is tested, more and more errors are identified and removed resulting in reduced failure rate. This error removal continues at a slower pace during the useful life of the product. As the software becomes obsolete no error corrections occurs and the failure rate remains unchanged.

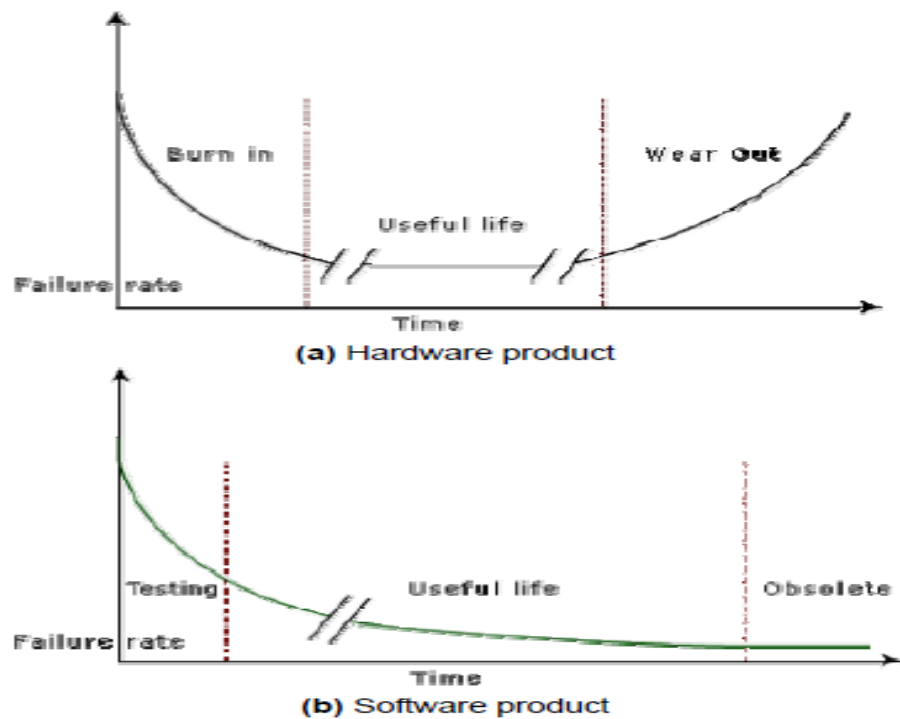


Fig. 13.1: Change in failure rate of a product

13.1.3 Reliability metrics

The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the SRS (software requirements specification) document. In order to be able to do this, some metrics are needed to quantitatively express the reliability of a software product. A good reliability measure should be observer-dependent, so that different people can agree on the degree of reliability a system has. For example, there are precise techniques for measuring performance, which would result in obtaining the same performance value irrespective of who is carrying out the performance measurement. However, in practice, it is very difficult to formulate a precise reliability measurement technique. The next base case is to have measures that correlate with reliability. There are six reliability metrics which can be used to quantify the reliability of software products.

- **Rate of occurrence of failure (ROCOF).** ROCOF measures the frequency of occurrence of unexpected behavior (i.e. failures). ROCOF measure of a software product can be obtained by observing the behavior

of a software product in operation over a specified time interval and then recording the total number of failures occurring during the interval.

- **Mean Time To Failure (MTTF).** MTTF is the average time between two successive failures, observed over a large number of failures. To measure MTTF, we can record the failure data for n failures. Let the failures occur at the time instants t_1, t_2, \dots, t_n . Then, MTTF can be calculated as

$$\sum_{i=1}^n \frac{t_{i+1} - t_i}{(n - 1)}$$

It is important to note that only run time is considered in the time measurements, i.e. the time for which the system is down to fix the error, the boot time, etc are not taken into account in the time measurements and the clock is stopped at these times.

- **Mean Time To Repair (MTTR).** Once failure occurs, some time is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.
- **Mean Time Between Failure (MTBR).** MTTF and MTTR can be combined to get the MTBR metric: $MTBF = MTTF + MTTR$. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours. In this case, time measurements are real time and not the execution time as in MTTF.
- **Probability of Failure on Demand (POFOD).** Unlike the other metrics discussed, this metric does not explicitly involve time measurements. POFOD measures the likelihood of the system failing when a service request is made. For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure.
- **Availability.** Availability of a system is a measure of how likely shall the system be available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs. This metric is important for systems such as telecommunication systems, and operating systems, which are supposed to

be never down and where repair and restart time are significant and loss of service during that time is important.

13.1.4 Classification of software failures

A possible classification of failures of software products into five different types is as follows:

- **Transient.** Transient failures occur only for certain input values while invoking a function of the system.
- **Permanent.** Permanent failures occur for all input values while invoking a function of the system.
- **Recoverable.** When recoverable failures occur, the system recovers with or without operator intervention.
- **Unrecoverable.** In unrecoverable failures, the system may need to be restarted.
- **Cosmetic.** These classes of failures cause only minor irritations, and do not lead to incorrect results. An example of a cosmetic failure is the case where the mouse button has to be clicked twice instead of once to invoke a given function through the graphical user interface.

13.1.5 Reliability growth models

A reliability growth model is a mathematical model of how software reliability improves as errors are detected and repaired. A reliability growth model can be used to predict when (or if at all) a particular level of reliability is likely to be attained. Thus, reliability growth modeling can be used to determine when to stop testing to attain a given reliability level. Although several different reliability growth models have been proposed, in this text we will discuss only two very simple reliability growth models.

13.1.5.1 Jelinski and Moranda Model

The simplest reliability growth model is a step function model where it is assumed that the reliability increases by a constant increment each time an error is detected and repaired. Such a model is shown in [fig. 13.2](#). However, this simple model of reliability which implicitly assumes that all errors contribute equally to reliability

growth, is highly unrealistic since it is already known that correction of different types of errors contribute differently to reliability growth.

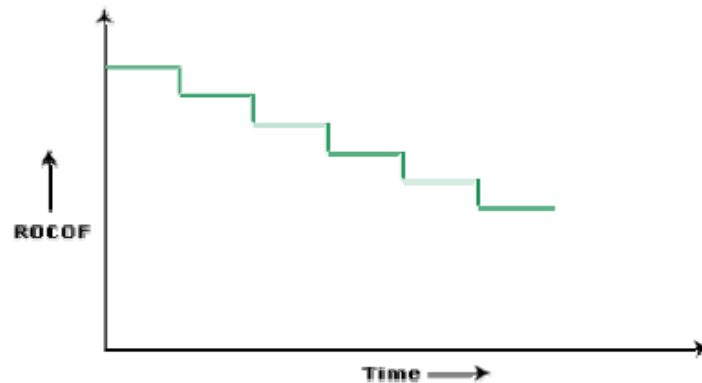


Fig. 13.2: Step function model of reliability growth

13.1.5.2 Littlewood and Verall's Model

This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors. It also models the fact that as errors are repaired, the average improvement in reliability per repair decreases (Fig. 13.3). It treats an error's contribution to reliability improvement to be an independent random variable having Gamma distribution. This distribution models the fact that error corrections with large contributions to reliability growth are removed first. This represents diminishing return as test continues.

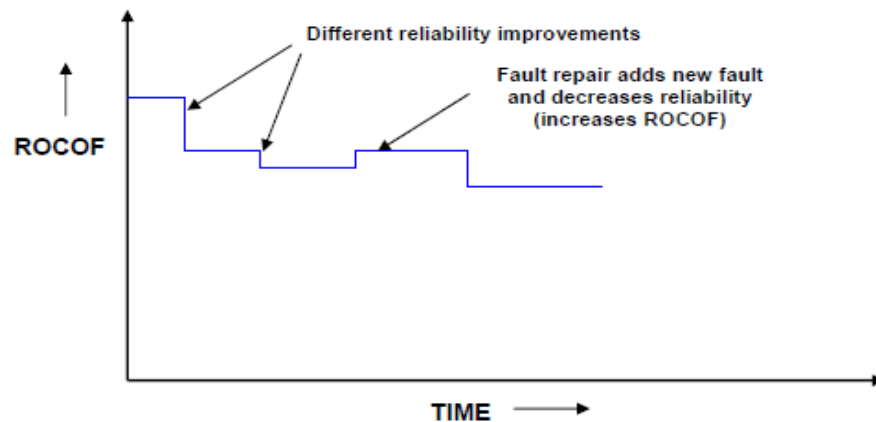


Fig. 13.3: Random-step function model of reliability growth

13.2 STATISTICAL TESTING AND SOFTWARE QUALITY MANAGEMENT

13.2.1 Statistical testing

Statistical testing is a testing process whose objective is to determine the reliability of software products rather than discovering errors. Test cases are designed for statistical testing with an entirely different objective than those of conventional testing.

13.2.1.1 Operation profile

Different categories of users may use a software for different purposes. For example, a Librarian might use the library automation software to create member records, add books to the library, etc. whereas a library member might use to software to query about the availability of the book, or to issue and return books. Formally, the operation profile of a software can be defined as the probability distribution of the input of an average user. If the input to a number of classes $\{C_i\}$ is divided, the probability value of a class represent the probability of an average user selecting his next input from this class. Thus, the operation profile assigns a probability value P_i to each input class C_i .

13.2.1.2 Steps in statistical testing

Statistical testing allows one to concentrate on testing those parts of the system that are most likely to be used. The first step of statistical testing is to determine the operation profile of the software. The next step is to generate a set of test data corresponding to the determined operation profile. The third step is to apply the test cases to the software and record the time between each failure. After a statistically significant number of failures have been observed, the reliability can be computed.

13.2.1.3 Advantages and disadvantages of statistical testing

Statistical testing allows one to concentrate on testing parts of the system that are most likely to be used. Therefore, it results in a system that the users to be more reliable (than actually it is!). Reliability estimation using statistical testing is more accurate compared to those of other methods such as ROCOF, POFOD etc. But it is not easy to perform statistical testing properly. There is no simple and repeatable way of defining operation profiles. Also, it is very much cumbersome

to generate test cases for statistical testing cause the number of test cases with which the system is to be tested should be statistically significant.

13.2.2 Software Quality

Traditionally, a quality product is defined in terms of its fitness of purpose. That is, a quality product does exactly what the users want it to do. For software products, fitness of purpose is usually interpreted in terms of satisfaction of the requirements laid down in the SRS document. Although “fitness of purpose” is a satisfactory definition of quality for many products such as a car, a table fan, a grinding machine, etc. – for software products, “fitness of purpose” is not a wholly satisfactory definition of quality. To give an example, consider a software product that is functionally correct. That is, it performs all functions as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally correct, we cannot consider it to be a quality product. Another example may be that of a product which does everything that the users want but has an almost incomprehensible and unmaintainable code. Therefore, the traditional concept of quality as “fitness of purpose” for software products is not wholly satisfactory.

The modern view of a quality associates with a software product several quality factors such as the following:

- **Portability:** A software product is said to be portable, if it can be easily made to work in different operating system environments, in different machines, with other software products, etc.
- **Usability:** A software product has good usability, if different categories of users (i.e. both expert and novice users) can easily invoke the functions of the product.
- **Reusability:** A software product has good reusability, if different modules of the product can easily be reused to develop new products.
- **Correctness:** A software product is correct, if different requirements as specified in the SRS document have been correctly implemented.
- **Maintainability:** A software product is maintainable, if errors can be easily corrected as and when they show up, new functions can be easily

added to the product, and the functionalities of the product can be easily modified, etc.

13.2.2.1 Software quality management system

A quality management system (often referred to as quality system) is the principal methodology used by organizations to ensure that the products they develop have the desired quality. A quality system consists of the following:

- **Managerial Structure and Individual Responsibilities.** A quality system is actually the responsibility of the organization as a whole. However, every organization has a separate quality department to perform several quality system activities. The quality system of an organization should have support of the top management. Without support for the quality system at a high level in a company, few members of staff will take the quality system seriously.
- **Quality System Activities.** The quality system activities encompass the following:
 - Auditing of projects
 - Review of the quality system
 - Development of standards, procedures, and guidelines, etc.
 - Production of reports for the top management summarizing the effectiveness of the quality system in the organization.

13.2.2.2 Evolution of quality management system

Quality systems have rapidly evolved over the last five decades. Prior to World War II, the usual method to produce quality products was to inspect the finished products to eliminate defective products. Since that time, quality systems of organizations have undergone through four stages of evolution as shown in the fig. 13.4. The initial product inspection method gave way to quality control (QC). Quality control focuses not only on detecting the defective products and eliminating them but also on determining the causes behind the defects. Thus, quality control aims at correcting the causes of errors and not just rejecting the products. The next breakthrough in quality systems was the development of quality assurance principles.

The basic premise of modern quality assurance is that if an organization's processes are good and are followed rigorously, then the products are bound to be of good quality. The modern quality paradigm includes guidance for recognizing, defining, analyzing, and improving the production process. Total quality management (TQM) advocates that the process followed by an organization must be continuously improved through process measurements.

TQM goes a step further than quality assurance and aims at continuous process improvement. TQM goes beyond documenting processes to optimizing them through redesign. A term related to TQM is Business Process Reengineering (BPR). BPR aims at reengineering the way business is carried out in an organization. From the above discussion it can be stated that over the years the quality paradigm has shifted from product assurance to process assurance (as shown in fig. 13.4).

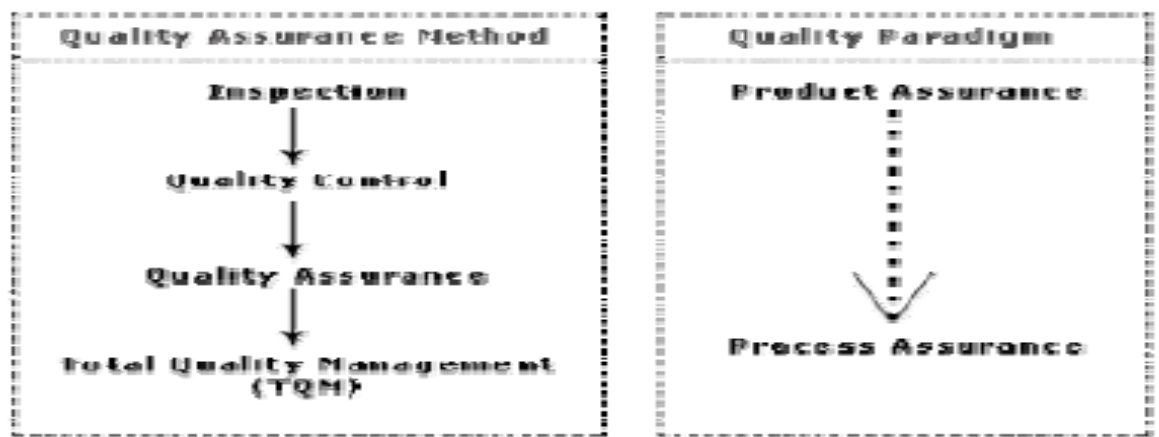


Fig. 13.4: Evolution of quality system and corresponding shift in the quality paradigm

13.3 ISO 9000

13.3.1 ISO 9000 certification

ISO (International Standards Organization) is a consortium of 63 countries established to formulate and foster standardization. ISO published its 9000 series of standards in 1987. ISO certification serves as a reference for contract between independent parties. The ISO 9000 standard specifies the guidelines for

maintaining a quality system. We have already seen that the quality system of an organization applies to all activities related to its product or service. The ISO standard mainly addresses operational aspects and organizational aspects such as responsibilities, reporting, etc. In a nutshell, ISO 9000 specifies a set of guidelines for repeatable and high-quality product development. It is important to realize that ISO 9000 standard is a set of guidelines for the production process and is not directly concerned about the product itself.

13.3.2 Types of ISO 9000 quality standards

ISO 9000 is a series of three standards: ISO 9001, ISO 9002, and ISO 9003. The ISO 9000 series of standards is based on the premise that if a proper process is followed for production, then good quality products are bound to follow automatically. The types of industries to which the different ISO standards apply are as follows.

ISO 9001 applies to the organizations engaged in design, development, production, and servicing of goods. This is the standard that is applicable to most software development organizations.

ISO 9002 applies to those organizations which do not design products but are only involved in production. Examples of these category industries include steel and car manufacturing industries that buy the product and plant designs from external sources and are involved in only manufacturing those products. Therefore, ISO 9002 is not applicable to software development organizations.

ISO 9003 applies to organizations that are involved only in installation and testing of the products.

13.3.3 Software products vs. other products

There are mainly two differences between software products and any other type of products.

- Software is intangible in nature and therefore difficult to control. It is very difficult to control and manage anything that is not seen. In contrast, any other industries such as car manufacturing industries where one can see a product being developed through various stages such as fitting engine, fitting doors, etc.

Therefore, it is easy to accurately determine how much work has been completed and to estimate how much more time will it take.

- During software development, the only raw material consumed is data. In contrast, large quantities of raw materials are consumed during the development of any other product.

13.3.4 Need for obtaining ISO 9000 certification

There is a mad scramble among software development organizations for obtaining ISO certification due to the benefits it offers. Some benefits that can be acquired to organizations by obtaining ISO certification are as follows:

- Confidence of customers in an organization increases when organization qualifies for ISO certification. This is especially true in the international market. In fact, many organizations awarding international software development contracts insist that the development organization have ISO 9000 certification. For this reason, it is vital for software organizations involved in software export to obtain ISO 9000 certification.
- ISO 9000 requires a well-documented software production process to be in place. A well-documented software production process contributes to repeatable and higher quality of the developed software.
- ISO 9000 makes the development process focused, efficient, and cost-effective.
- ISO 9000 certification points out the weak points of an organization and recommends remedial action.
- ISO 9000 sets the basic framework for the development of an optimal process and Total Quality Management (TQM).

13.3.5 SUMMARY of ISO 9001 certification

A SUMMARY of the main requirements of ISO 9001 as they relate of software development is as follows. Section numbers in brackets correspond to those in the standard itself:

Management Responsibility (4.1)

- The management must have an effective quality policy.

- The responsibility and authority of all those whose work affects quality must be defined and documented.
- A management representative, independent of the development process, must be responsible for the quality system. This requirement probably has been put down so that the person responsible for the quality system can work in an unbiased manner.
- The effectiveness of the quality system must be periodically reviewed by audits.

Quality System (4.2)

A quality system must be maintained and documented.

Contract Reviews (4.3)

Before entering into a contract, an organization must review the contract to ensure that it is understood, and that the organization has the necessary capability for carrying out its obligations.

Design Control (4.4)

- The design process must be properly controlled, this includes controlling coding also. This requirement means that a good configuration control system must be in place.
- Design inputs must be verified as adequate.
- Design must be verified.
- Design output must be of required quality.
- Design changes must be controlled.

Document Control (4.5)

- There must be proper procedures for document approval, issue and removal.
- Document changes must be controlled. Thus, use of some configuration management tools is necessary.

Purchasing (4.6)

Purchasing material, including bought-in software must be checked for conforming to requirements.

Purchaser Supplied Product (4.7)

Material supplied by a purchaser, for example, client-provided software must be properly managed and checked.

Product Identification (4.8)

The product must be identifiable at all stages of the process. In software terms this means configuration management.

Process Control (4.9)

- The development must be properly managed.
- Quality requirement must be identified in a quality plan.

Inspection and Testing (4.10)

In software terms this requires effective testing i.e., unit testing, integration testing and system testing. Test records must be maintained.

Inspection, Measuring and Test Equipment (4.11)

If integration, measuring, and test equipment are used, they must be properly maintained and calibrated.

Inspection and Test Status (4.12)

The status of an item must be identified. In software terms this implies configuration management and release control.

Control of Nonconforming Product (4.13)

In software terms, this means keeping untested or faulty software out of the released product, or other places where it might cause damage.

Corrective Action (4.14)

This requirement is both about correcting errors when found, and also investigating why the errors occurred and improving the process to prevent occurrences. If an error occurs despite the quality system, the system needs improvement.

Handling, (4.15)

This clause deals with the storage, packing, and delivery of the software product.

Quality records (4.16)

Recording the steps taken to control the quality of the process is essential in order to be able to confirm that they have actually taken place.

Quality Audits (4.17)

Audits of the quality system must be carried out to ensure that it is effective.

Training (4.18)

Training needs must be identified and met.

13.3.6 Salient features of ISO 9001 certification

The salient features of ISO 9001 are as follows:

- All documents concerned with the development of a software product should be properly managed, authorized, and controlled. This requires a configuration management system to be in place.
- Proper plans should be prepared and then progress against these plans should be monitored.
- Important documents should be independently checked and reviewed for effectiveness and correctness.
- The product should be tested against specification.
- Several organizational aspects should be addressed e.g., management reporting of the quality team.

13.3.7 Shortcomings of ISO 9000 certification

Even though ISO 9000 aims at setting up an effective quality system in an organization, it suffers from several shortcomings. Some of these shortcomings of the ISO 9000 certification process are the following:

- ISO 9000 requires a software production process to be adhered to but does not guarantee the process to be of high quality. It also does not give any guideline for defining an appropriate process.
- ISO 9000 certification process is not fool-proof and no international accreditation agency exists. Therefore, it is likely that variations in the norms of awarding certificates can exist among the different accreditation agencies and also among the registrars.
- Organizations getting ISO 9000 certification often tend to downplay domain expertise. These organizations start to believe that since a good process is in place, any engineer is as effective as any other engineer in doing any particular activity relating to software development. However, many areas of software development are so specialized that special expertise and experience in these areas (domain expertise) is required. In manufacturing industry there is a clear link between process quality and

product quality. Once a process is calibrated, it can be run again and again producing quality goods. In contrast, software development is a creative process and individual skills and experience are important.

- ISO 9000 does not automatically lead to continuous process improvement, i.e. does not automatically lead to TQM.

13.4 SEI CMM

13.4.1 SEI Capability Maturity Model

SEI Capability Maturity Model (SEI CMM) helped organizations to improve the quality of the software they develop and therefore adoption of SEI CMM model has significant business benefits.

SEI CMM can be used two ways: capability evaluation and software process assessment. Capability evaluation and software process assessment differ in motivation, objective, and the final use of the result. Capability evaluation provides a way to assess the software process capability of an organization. The results of capability evaluation indicates the likely contractor performance if the contractor is awarded a work. Therefore, the results of software process capability assessment can be used to select a contractor. On the other hand, software process assessment is used by an organization with the objective to improve its process capability. Thus, this type of assessment is for purely internal use.

SEI CMM classifies software development industries into the following five maturity levels. The different levels of SEI CMM have been designed so that it is easy for an organization to slowly build its quality system starting from scratch.

Level 1: Initial. A software development organization at this level is characterized by ad hoc activities. Very few or no processes are defined and followed. Since software production processes are not defined, different engineers follow their own process and as a result, development efforts become chaotic. Therefore, it is also called chaotic level. The success of projects depends on individual efforts and heroics. When engineers leave, the successors have great difficulty in understanding the process followed and the work completed. Since

formal project management practices are not followed, under time pressure short cuts are tried out leading to low quality.

Level 2: Repeatable. At this level, the basic project management practices such as tracking cost and schedule are established. Size and cost estimation techniques like function point analysis, COCOMO, etc. are used. The necessary process discipline is in place to repeat earlier success on projects with similar applications. Please remember that opportunity to repeat a process exists only when a company produces a family of products.

Level 3: Defined. At this level the processes for both management and development activities are defined and documented. There is a common organization-wide understanding of activities, roles, and responsibilities. The processes though defined, the process and product qualities are not measured. ISO 9000 aims at achieving this level.

Level 4: Managed. At this level, the focus is on software metrics. Two types of metrics are collected. Product metrics measure the characteristics of the product being developed, such as its size, reliability, time complexity, understandability, etc. Process metrics reflect the effectiveness of the process being used, such as average defect correction time, productivity, average number of defects found per hour inspection, average number of failures detected during testing per LOC, etc. Quantitative quality goals are set for the products. The software process and product quality are measured and quantitative quality requirements for the product are met. Various tools like Pareto charts, fishbone diagrams, etc. are used to measure the product and process quality. The process metrics are used to check if a project performed satisfactorily. Thus, the results of process measurements are used to evaluate project performance rather than improve the process.

Level 5: Optimizing. At this stage, process and product metrics are collected. Process and product measurement data are analyzed for continuous process improvement. For example, if from an analysis of the process measurement results, it was found that the code reviews were not very effective and a large number of errors were detected only during the unit testing, then the process may be fine tuned to make the review more effective. Also, the lessons learned from specific projects are incorporated in to the process. Continuous process improvement is achieved both by carefully analyzing the quantitative feedback

from the process measurements and also from application of innovative ideas and technologies. Such an organization identifies the best software engineering practices and innovations which may be tools, methods, or processes. These best practices are transferred throughout the organization.

Key process areas (KPA) of a software organization

Except for SEI CMM level 1, each maturity level is characterized by several Key Process Areas (KPAs) that includes the areas an organization should focus to improve its software process to the next level. The focus of each level and the corresponding key process areas are shown in the fig. 13.5.

CMM Level	Focus	Key Process Ares
1. Initial	Competent people	
2. Repeatable	Project management	Software project planning Software configuration management
3. Defined	Definition of processes	Process definition Training program Peer reviews
4. Managed	Product and process quality	Quantitative process metrics Software quality management
5. Optimizing	Continuous process improvement	Defect prevention Process change management Technology change management

Fig. 13.5: The focus of each SEI CMM level and the corresponding key process areas

SEI CMM provides a list of key areas on which to focus to take an organization from one level of maturity to the next. Thus, it provides a way for gradual quality improvement over several stages. Each stage has been carefully designed such that one stage enhances the capability already built up. For example, it considers that trying to implement a defined process (SEI CMM level 3) before a repeatable process (SEI CMM level 2) would be counterproductive as it becomes difficult to follow the defined process due to schedule and budget pressures.

ISO 9000 certification vs. SEI/CMM

For quality appraisal of a software development organization, the characteristics of ISO 9000 certification and the SEI CMM differ in some respects. The differences are as follows:

- ISO 9000 is awarded by an international standards body. Therefore, ISO 9000 certification can be quoted by an organization in official documents, communication with external parties, and the tender quotations. However, SEI CMM assessment is purely for internal use.
- SEI CMM was developed specifically for software industry and therefore addresses many issues which are specific to software industry alone.
- SEI CMM goes beyond quality assurance and prepares an organization to ultimately achieve Total Quality Management (TQM). In fact, ISO 9001 aims at level 3 of SEI CMM model.
- SEI CMM model provides a list of key process areas (KPA) on which an organization at any maturity level needs to concentrate to take it from one maturity level to the next. Thus, it provides a way for achieving gradual quality improvement.

Applicability of SEI CMM to organizations

Highly systematic and measured approach to software development suits large organizations dealing with negotiated software, safety-critical software, etc. For those large organizations, SEI CMM model is perfectly applicable. But small organizations typically handle applications such as Internet, e-commerce, and are without an established product range, revenue base, and experience on past projects, etc. For such organizations, a CMM-based appraisal is probably excessive. These organizations need to operate more efficiently at the lower levels of maturity. For example, they need to practice effective project management, reviews, configuration management, etc.

Personal software process

Personal Software Process (PSP) is a scaled down version of the industrial software process. PSP is suitable for individual use. It is important to note that SEI CMM does not tell software developers how to analyze, design, code, test, or document software products, but assumes that engineers use effective personal

practices. PSP recognizes that the process for individual use is different from that necessary for a team.

The quality and productivity of an engineer is to a great extent dependent on his process. PSP is a framework that helps engineers to measure and improve the way they work. It helps in developing personal skills and methods by estimating and planning, by showing how to track performance against plans, and provides a defined process which can be tuned by individuals.

Time measurement. PSP advocates that engineers should track the way they spend time. Because, boring activities seem longer than actual and interesting activities seem short. Therefore, the actual time spent on a task should be measured with the help of a stop-clock to get an objective picture of the time spent. For example, he may stop the clock when attending a telephone call, taking a coffee break etc. An engineer should measure the time he spends for designing, writing code, testing, etc.

PSP Planning. Individuals must plan their project. They must estimate the maximum, minimum, and the average LOC required for the product. They should use their productivity in minutes/LOC to calculate the maximum, minimum, and the average development time. They must record the plan data in a project plan SUMMARY. The PSP is schematically shown in fig. 13.6. While carrying out the different phases, they must record the log data using time measurement. During post-mortem, they can compare the log data with their project plan to achieve better planning in the future projects, to improve their process, etc.

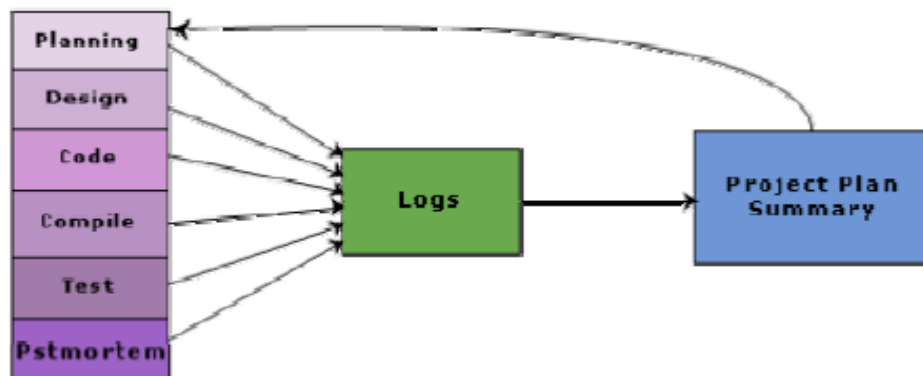


Fig. 13.6: Schematic representation of PSP

The PSP levels are summarized in fig. 13.7. PSP2 introduces defect management via the use of checklists for code and design reviews. The checklists are developed from gathering and analyzing defect data earlier projects.

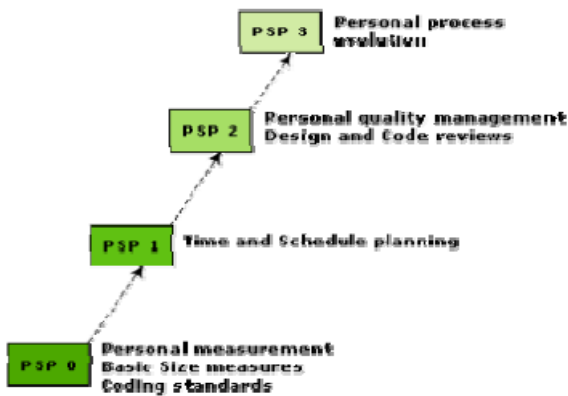


Fig. 13.7: Levels of PSP

13.4.2 Six Sigma

The purpose of Six Sigma is to improve processes to do things better, faster, and at lower cost. It can be used to improve every facet of business, from production, to human resources, to order entry, to technical support. Six Sigma can be used for any activity that is concerned with cost, timeliness, and quality of results. Therefore, it is applicable to virtually every industry.

Six Sigma at many organizations simply means striving for near perfection. Six Sigma is a disciplined, data-driven approach to eliminate defects in any process – from manufacturing to transactional and product to service.

The statistical representation of Six Sigma describes quantitatively how a process is performing. To achieve Six Sigma, a process must not produce more than 3.4 defects per million opportunities. A Six Sigma defect is defined as any system behavior that is not as per customer specifications. Total number of Six Sigma opportunities is then the total number of chances for a defect. Process sigma can easily be calculated using a Six Sigma calculator.

The fundamental objective of the Six Sigma methodology is the implementation of a measurement-based strategy that focuses on process improvement and variation reduction through the application of Six Sigma improvement projects. This is accomplished through the use of two Six Sigma sub-methodologies: DMAIC and DMADV. The Six Sigma DMAIC process (define, measure, analyze, improve,

control) is an improvement system for existing processes failing below specification and looking for incremental improvement. The Six Sigma DMADV process (define, measure, analyze, design, verify) is an improvement system used to develop new processes or products at Six Sigma quality levels. It can also be employed if a current process requires more than just incremental improvement. Both Six Sigma processes are executed by Six Sigma Green Belts and Six Sigma Black Belts, and are overseen by Six Sigma Master Black Belts.

Many frameworks exist for implementing the Six Sigma methodology. Six Sigma Consultants all over the world have also developed proprietary methodologies for implementing Six Sigma quality, based on the similar change management philosophies and applications of tools.

13.5 MODEL QUESTIONS

The following questions have been designed to test the objectives identified for this module:

1. Explain the relative advantages of repeatable and non-repeatable software development organization.
2. How can the reliability of a software product be increased?
3. Identify the factors which make the measurement of software reliability much harder problem than the measurement of hardware reliability.
4. Discuss how the reliability changes over the life time of a software product and a hardware product.
5. Define six metrics to measure software reliability. Do you consider these metrics entirely satisfactory to provide measure of the reliability of a system? Justify your answer.
6. Explain using one simple sentence each what you understand by the following reliability measures:
 - a. A POFOD of 0.001
 - b. A ROCOF of 0.002
 - c. MTBF of 200 units
 - d. Availability of 0.998
7. Differentiate among the characteristics of different types of failures of software products.

8. In the context of reliability growth modeling, compare the characteristics between Jelinski and Moranda Model, and Littlewood and Verall's Model.
9. What is the main objective of statistical testing and also identify the three specific steps of statistical testing.
10. Compare the relative advantages and disadvantages of statistical testing.
11. What does the quality parameter "fitness of purpose" mean in the context of software products? Why is this not a satisfactory criterion for determining the quality of software products?
12. What according to you is a quality software product?
13. If an organization does not document its quality system, what problems would it face?
14. In a software development organization, identify the persons responsible for carrying out the quality assurance activities. Explain the principal tasks they perform to meet this responsibility.
15. In a software development organization whose responsibility is to ensure that the products are of high quality? Explain the principal tasks they perform to meet this responsibility.
16. What do you understand by Total Quality Management (TQM)? What are the advantages of TQM? Does ISO 9000 standard aim for TQM?
17. What are the principal activities of a modern quality system?
18. What is meant by ISO 9000 certification?
19. Discuss the types of organizations to which different types of ISO standards are applicable.
20. Compare the characteristics of software products and other types of products.
21. Why is it important for a software development organization to obtain ISO 9001 certification?
22. List five salient requirements that a software development organization must comply with before it can be awarded the ISO 9001 certificate.
23. What are the salient features of ISO 9001 certification?
24. What are the shortcomings of ISO 9000 certification process?

25. What is the main purpose of SEI Capability Maturity Model (SEI CMM)?
How can SEI CMM model be used to improve the quality of software products?
26. Explain five different levels of SEI CMM model.
27. What do you understand by repeatable software development?
Organizations assessed at which level SEI CMM maturity to achieve repeatable software development?
28. Suppose an organization mentions in its job advertisement that it has been assessed at level 3 of SEI CMM, what can you infer the about the current quality practices at the organization? What does this organization have to do to reach SEI CMM level 4?
29. What is the difference between process metrics and product metrics? Give four examples of each.
30. Suppose you want to buy a certain software product and you have kept a purchase precondition that the vendor must install the software, train your manpower on that, and maintain the product for at least one year, only then you would release the payment. Also, you do not foresee any maintenance requirement for the product once it works satisfactorily. Now, you receive bids from three vendors. Two of the vendors quote Rs. 3 Lakhs and Rs. 4 Lakhs whereas the third vendor quotes Rs. 10 Lakhs saying that the prices would be high because they would be following a good development process as they have been assessed at the Level 5 of SEI CMM. Discuss how would you decide whom to award the contract.
31. What do you understand by Key Process Area (KPA), in the context of SEI CMM? Would there be any problem if an organization tries to implement higher level SEI CMM KPAs before achieving lower level KPAs? Justify your answer using suitable examples.
32. Compare the relative advantages and disadvantages of ISO 9001 certification and the SEI CMM-based quality assessment.
33. What do you mean by Personal Software Process (PSP)?
34. What is the Six Sigma quality initiative? To which category of industries is it applicable? Explain the Six Sigma technique adopted by software organization with respect to the goal, the procedure, and the outcome.

Mark all options which are true.

1. Repeatable software development implies which of the following?
 - a. software development process is person-dependent
 - b. software development process is person-independent
 - c. either software development process is person-dependent or person-independent
 - d. neither software development process is person-dependent nor person-independent
2. A type of failures that occurs for all input values while invoking a function of the system is
 - a. transient failure
 - b. permanent failure
 - c. recoverable failure
 - d. unrecoverable failure
3. The reliability growth modeling can be used
 - a. to improve the reliability of a software product as errors are detected and repaired
 - b. to predict when a particular level of reliability is likely to be attained
 - c. to determine when to stop testing to attain a given reliability level
 - d. all of the above
4. Statistical testing is based on first determining
 - a. operation profile
 - b. user profile
 - c. product profile
 - d. development process profile
5. The quality system activities encompass
 - a. auditing of projects
 - b. review of the quality system
 - c. development of standards, procedures, and guidelines, etc.
 - d. all of the above
6. The basic premise of modern quality assurance is
 - a. continuous process improvement

- b. thorough product testing
 - c. if an organization's processes are good and are followed rigorously then the products are bound to be of good quality
 - d. collection of process metrics
- 7.** Continuous process improvement is achieved through which stages of a quality system?
- a. quality control
 - b. quality assurance
 - c. total quality management
 - d. none of the above
- 8.** Which ISO 9000 standard can be applied to organizations engaged in design, development, production, and servicing of goods etc.?
- a. ISO 9001
 - b. ISO 9002
 - c. ISO 9003
 - d. none of the above
- 9.** Salient feature/features of ISO 9001 certification is/are
- a. all documents concerned with the development of a software product should be properly managed, authorized, and controlled
 - b. proper plans should be prepared and then progress against these plans should be monitored
 - c. the product should be tested against specification
 - d. all of the above
- 10.** In which level of SEI Capability Maturity Model the processes for both management and development activities are defined and documented?
- a. initial level
 - b. defined level
 - c. repeatable level
 - d. managed level
- 11.** In which level of SEI Capability Maturity Model both product and process metrics are defined?
- a. initial level
 - b. defined level

- c. repeatable level
- d. optimizing level

12. Continuous process improvement is achieved in which level of SEI Capability Maturity Model?

- a. initial level
- b. defined level
- c. repeatable level
- d. optimizing level
- e. managed level

13. Personal Software Process (PSP) is targeted for

- a. individual use
- b. team use
- c. individual use as well as team use
- d. none of the above

14. The purpose of Six Sigma is

- a. to improve development processes to do things better
- b. to improve development processes
- c. to make development processes cost effective
- d. all of the above

Mark the following as either True or False. Justify your answer.

1. Reliability of a software product is observer-independent.
2. The reliability of a software product increases almost linearly, each time a defect gets detected and fixed.
3. Reliability of a software product depends upon the product's execution profile.
4. Reliability behavior for hardware and software are almost same.
5. The reliability with time of a particular software product always increases.
6. As testing continues, the rate of growth of reliability slows down representing a diminishing return of reliability growth with testing effort.
7. The term "fitness of purpose" is appropriate for defining a quality software product.

8. Modern quality assurance paradigms are centered around carrying out through product testing.
9. One of the major criteria for obtaining ISO 9001 certification for a software development organization is to possess well-documented software production process.
10. One of the uses of receiving ISO 9001 certification by a software organization is that it can improve its sales efforts by advertising its products as conforming to ISO 9001 certification.
11. ISO 9000 gives specific guidelines for defining an appropriate process for the development of a particular product in an organization.

UNIT XIV: SOFTWARE MAINTENANCE

14.0 LEARNING OBJECTIVES

At the end of this lesson the student would be able to:

- Explain the necessity of software maintenance.
- Identify the types of software maintenance.
- Identify the disadvantages associated with software maintenance.
- Explain what is meant by software reverse engineering.
- What are legacy software products? Identify the problems in their maintenance.
- Identify the factors upon which software maintenance activities depend.
- Identify the process models for software maintenance.
- Explain what is meant by software reengineering.
- Estimate the approximate maintenance cost of a software product.

14.1 CHARACTERISTICS OF SOFTWARE MAINTENANCE

14.1.1 Necessity of software maintenance

Software maintenance is becoming an important activity of a large number of software organizations. This is no surprise, given the rate of hardware obsolescence, the immortality of a software product per se, and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features. When the hardware platform is changed, and a software product performs some low-level functions, maintenance is necessary. Also, whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface. For instance, a software product may need to be maintained when the operating system changes. Thus, every software product continues to evolve after

its development through maintenance efforts. Therefore, it can be stated that software maintenance is needed to correct errors, enhance features, port the software to new platforms, etc.

14.1.2 Types of software maintenance

There are basically three types of software maintenance. These are:

- **Corrective:** Corrective maintenance of a software product is necessary to rectify the bugs observed while the system is in use.
- **Adaptive:** A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.
- **Perfective:** A software product needs maintenance to support the new features that users want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

14.2 PROBLEMS ASSOCIATED WITH SOFTWARE MAINTENANCE

Software maintenance work typically is much more expensive than what it should be and takes more time than required. In software organizations, maintenance work is mostly carried out using ad hoc techniques. The primary reason being that software maintenance is one of the most neglected areas of software engineering. Even though software maintenance is fast becoming an important area of work for many companies as the software products of yester years age, still software maintenance is mostly being carried out as fire-fighting operations, rather than through systematic and planned activities.

Software maintenance has a very poor image in industry. Therefore, an organization often cannot employ bright engineers to carry out maintenance work. Even though maintenance suffers from a poor image, the work involved is often more challenging than development work. During maintenance it is necessary to thoroughly understand someone else's work and then carry out the required modifications and extensions.

Another problem associated with maintenance work is that the majority of software products needing maintenance are legacy products.

14.3 SOFTWARE REVERSE ENGINEERING

Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code. The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system. Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured. Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.

The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its readability, structure, and understandability, without changing of its functionalities. A process model for reverse engineering has been shown in fig. 14.1. A program can be reformatted using any of the several available pretty printer programs which layout the program neatly. Many legacy software products with complex control structure and unthoughtful variable names are difficult to comprehend. Assigning meaningful variable names is important because meaningful variable names are the most helpful thing in code documentation. All variables, data structures, and functions should be assigned meaningful names wherever possible. Complex nested conditionals in the program can be replaced by simpler conditional statements or whenever appropriate by case statements.

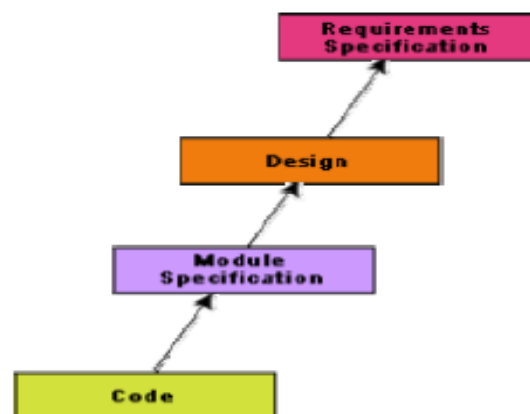


Fig. 14.1: A process model for reverse engineering

After the cosmetic changes have been carried out on a legacy software, the process of extracting the code, design, and the requirements specification can begin. These activities are schematically shown in fig. 14.2. In order to extract the design, a full understanding of the code is needed. Some automatic tools can be used to derive the data flow and control flow diagram from the code. The structure chart (module invocation sequence and data interchange among modules) should also be extracted. The SRS document can be written once the full code has been thoroughly understood and the design extracted.

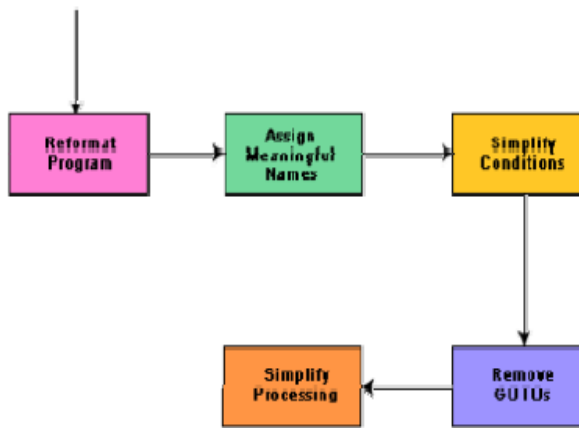


Fig. 14.2: Cosmetic changes carried out before reverse engineering

14.4 LEGACY SOFTWARE PRODUCTS

It is prudent to define a legacy system as any software system that is hard to maintain. The typical problems associated with legacy systems are poor documentation, unstructured (spaghetti code with ugly control structure), and lack of personnel knowledgeable in the product. Many of the legacy systems were developed long time back. But, it is possible that a recently developed system having poor design and documentation can be considered to be a legacy system.

Check Your Progress I

Mark all options which are true.

1. Software products need maintenance to
 - a. correct errors

- b. enhance features
 - c. port to new platforms
 - d. overcome wear and tear caused by use
2. Software products need adaptive maintenance for which of the following reasons?
- a. to rectify bugs observed while the system is in use
 - b. when the customers need the product to run on new platforms.
 - c. to support the new features that users want it to support.
 - d. all of the above
3. Hardware products need maintenance to
- a. correct errors
 - b. enhance features
 - c. port to new platforms
 - d. overcome wear and tear caused by use
4. Legacy software products having poor design and code structure are maintained by performing which task?
- a. the code can be directly modified and the changes appropriately reflected in all the relevant documents
 - b. suitable software maintenance process must be followed by a reverse engineering cycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents
 - c. none of the above

Mark the following statements as either True or False.

- 5. Corrective maintenance is the type of maintenance that is frequently carried out on average software product.
- 6. Only badly designed software products need maintenance.
- 7. The structure of a program may be degraded as more and more maintenance is carried out.
- 8. Legacy software products are very difficult to maintain.
- 9. Legacy products are those products which have been developed long time back.

14.5 FACTORS ON WHICH SOFTWARE MAINTENANCE ACTIVITIES DEPENDS

The activities involved in a software maintenance project are not unique and depend on several factors such as:

- The extent of modification to the product required
- The resources available to the maintenance team
- The conditions of the existing product (e.g., how structured it is, how well documented it is, etc.)
- The expected project risks, etc.

When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately reflected in all the documents. But more elaborate activities are required when the required changes are not so trivial. Usually, for complex maintenance projects for legacy systems, the software process can be represented by a reverse engineering cycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents.

14.6 SOFTWARE MAINTENANCE PROCESS MODELS

Two broad categories of process models for software maintenance can be proposed. The first model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later. This maintenance process is graphically presented in fig. 14.3. In this approach, the project starts by gathering the requirements for changes. The requirements are next analyzed to formulate the strategies to be adopted for code change. At this stage, the association of at least a few members of the original development team goes a long way in reducing the cycle time, especially for projects involving unstructured and inadequately documented code. The availability of a working old system to the maintenance engineers at the maintenance site greatly facilitates the task of the maintenance team as they get a good insight into the working of the old system and also can compare the working of their modified system with the old system. Also, debugging of the reengineered

system becomes easier as the program traces of both the systems can be compared to localize the bugs.

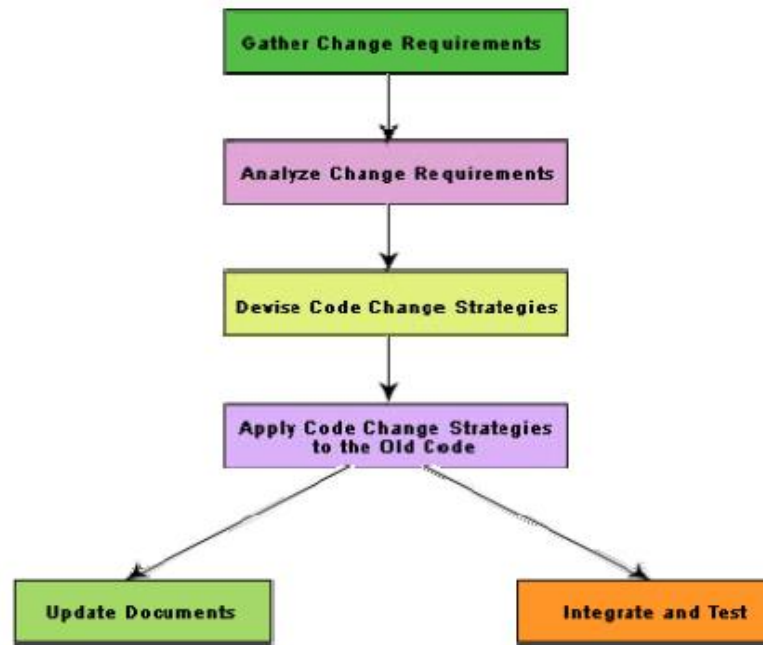


Fig. 14.3: Maintenance process model 1

The second process model for software maintenance is preferred for projects where the amount of rework required is significant. This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an approach is also known as software reengineering. This process model is depicted in fig. 14.4. The reverse engineering cycle is required for legacy products. During the reverse engineering, the old code is analyzed (abstracted) to extract the module specifications. The module specifications are then analyzed to produce the design. The design is analyzed (abstracted) to produce the original requirements specification. The change requests are then applied to this requirements specification to arrive at the new requirements specification. At the design, module specification, and coding a substantial reuse is made from the reverse engineered products. An important advantage of this approach is that it produces a more structured design compared to what the original product had, produces good documentation, and very often results in increased efficiency. The efficiency improvements are brought about by a more efficient design. However, this approach is more costly than the first approach. An empirical study indicates that process 1 is preferable when the amount of rework is no more than 15% (as

shown in fig. 14.5). Besides the amount of rework, several other factors might affect the decision regarding using process model 1 over process model 2:

- Reengineering might be preferable for products which exhibit a high failure rate.
- Reengineering might also be preferable for legacy products having poor design and code structure.

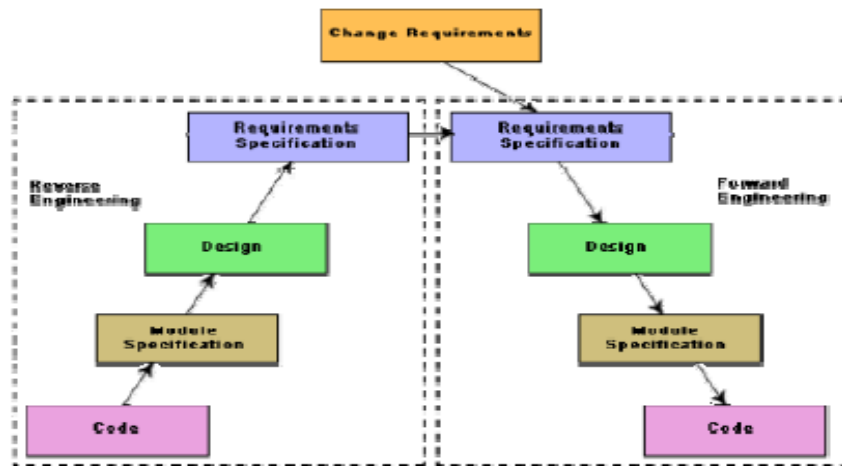


Fig. 14.4: Maintenance process model 2

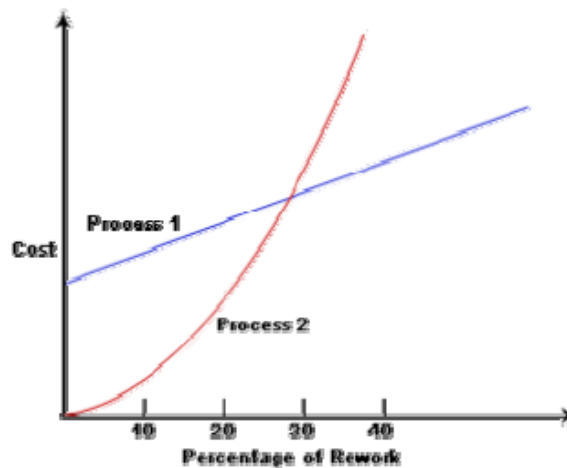


Fig. 14.5: Empirical estimation of maintenance cost versus percentage rework

14.7 SOFTWARE REENGINEERING

Software reengineering is a combination of two consecutive processes i.e. software reverse engineering and software forward engineering as shown in the fig. 14.4.

14.8 ESTIMATION OF APPROXIMATE MAINTENANCE COST

It is well known that maintenance efforts require about 60% of the total life cycle cost for a typical software product. However, maintenance costs vary widely from one application domain to another. For embedded systems, the maintenance cost can be as much as 2 to 4 times the development cost.

Boehm [1981] proposed a formula for estimating maintenance costs as part of his COCOMO cost estimation model. Boehm's maintenance cost estimation is made in terms of a quantity called the Annual Change Traffic (ACT). Boehm defined ACT as the fraction of a software product's source instructions which undergo change during a typical year either through addition or deletion.

$$ACT = \frac{KLOC_{added} + KLOC_{deleted}}{KLOC_{total}}$$

where, $KLOC_{added}$ is the total kilo lines of source code added during maintenance. $KLOC_{deleted}$ is the total KLOC deleted during maintenance. Thus, the code that is changed, should be counted in both the code added and the code deleted. The annual change traffic (ACT) is multiplied with the total development cost to arrive at the maintenance cost:

$$\text{maintenance cost} = ACT \times \text{development cost}$$

Most maintenance cost estimation models, however, yield only approximate results because they do not take into account several factors such as experience level of the engineers, and familiarity of the engineers with the product, hardware requirements, software complexity, etc.

Check Your Progress II

Mark all options which are true.

1. A reverse engineering cycle during maintenance phase is required for which type of software products?
 - a. well documented software products
 - b. well-structured software products
 - c. legacy software products
 - d. both well documented and well-structured software products

2. Reengineering is preferable for which of the software products?
 - a. software products exhibiting high failure rates
 - b. software products having poor design
 - c. software products having poor code structure
 - d. all of the above
3. Identify which of the following factors software maintenance cost estimation models should take into account.
 - a. experience level of the engineers
 - b. familiarity of the engineers with the product
 - c. hardware requirements
 - d. software complexity
 - e. all of the above
4. Software maintenance effort requires approximately what percentage of the total life cycle cost for a typical software product?
 - a. about 90%
 - b. about 70%
 - c. about 60%
 - d. about 40%

Mark the following statements as either True or False.

5. In the process of reverse engineering, we change the functionalities of an existing code.

14.9 MODEL QUESTIONS

1. What for software products are required to maintain?
2. What are the different types of maintenance that a software product might need? Why are these maintenances required?
3. What are the disadvantages associated with software maintenance?
4. What do you mean by the term software reverse engineering? Why is it required? Explain the different activities undertaken during reverse engineering.
5. What is legacy software product? Explain the problems one would encounter while maintaining a legacy product.

6. What are the different factors upon which software maintenance activities depend?
7. What do you mean by the term software reengineering? Why is it required?
8. If the development cost of a software product is Rs. 10,000,000/-, compute the annual maintenance cost given that every year approximately 5% of the code needs modification. Identify the factors which render the maintenance cost estimation inaccurate.

UNIT XV: COMPUTER AIDED SOFTWARE ENGINEERING

15.0 LEARNING OBJECTIVES

At the end of this lesson the student would be able to:

- What is meant by CASE tool?
- Identify the primary reasons for using a CASE tool.
- What is meant by a CASE environment?
- Differentiate in between a CASE environment and a programming environment.
- Identify the benefits of a CASE environment.
- Identify the features of a prototyping CASE tool.
- Identify the features that a good prototyping CASE tool should support.
- Identify the supports that are typically available from CASE tools in order to perform structured analysis and software design activity.
- Identify the support that might be available from CASE tools during code generation.
- Identify the features of a test case generation CASE tool.
- Identify hardware and environmental requirements for a CASE tool.
- Identify the software documentation support that might be available from CASE tools.
- Identify the two project management supports that should be available from CASE tools.
- Identify the support that needs to be provided by CASE tools for interfacing with other CASE tools.
- Identify the two software reverse engineering supports that should be available from CASE tools.
- Identify the two features that might be supported by data dictionary interface of a CASE environment.

- Identify the non-traditional features supported by second generation CASE tools.
- Explain the architecture of a CASE environment with the help of a suitable diagram.

15.1 BASIC IDEAS ON CASE TOOLS

15.1.1 CASE tool and its scope

A CASE (Computer Aided Software Engineering) tool is a generic term used to denote any form of automated support for software engineering. In a more restrictive sense, a CASE tool means any tool used to automate some activity associated with software development. Many CASE tools are available. Some of these CASE tools assist in phase related tasks such as specification, structured analysis, design, coding, testing, etc.; and others to non-phase activities such as project management and configuration management.

15.1.1.1 Reasons for using CASE tools

The primary reasons for using a CASE tool are:

- To increase productivity
- To help produce better quality software at lower cost

15.1.1.2 CASE environment

Although individual CASE tools are useful, the true power of a tool set can be realized only when these set of tools are integrated into a common framework or environment. CASE tools are characterized by the stage or stages of software development life cycle on which they focus. Since different tools covering different stages share common information, it is required that they integrate through some central repository to have a consistent view of information associated with the software development artifacts. This central repository is usually a data dictionary containing the definition of all composite and elementary data items. Through the central repository all the CASE tools in a CASE environment share common information among themselves. Thus a CASE environment facilitates the automation of the step-by-step methodologies for

software development. A schematic representation of a CASE environment is shown in fig. 15.1.

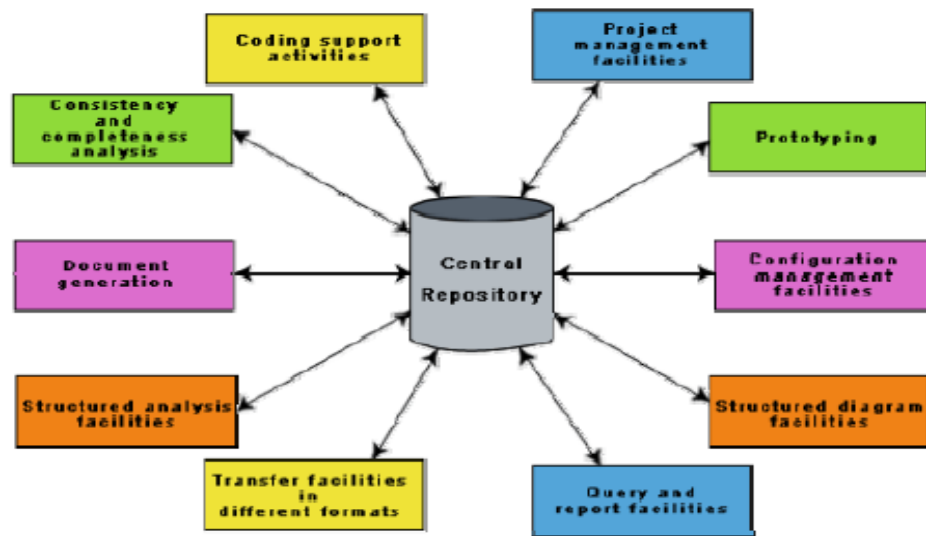


Fig. 15.1: A CASE Environment

15.1.1.3 CASE environment vs programming environment

A CASE environment facilitates the automation of the step-by-step methodologies for software development. In contrast to a CASE environment, a programming environment is an integrated collection of tools to support only the coding phase of software development.

15.1.1.4 Benefits of CASE

Several benefits accrue from the use of a CASE environment or even isolated CASE tools. Some of those benefits are:

- A key benefit arising out of the use of a CASE environment is cost saving through all development phases. Different studies carry out to measure the impact of CASE put the effort reduction between 30% to 40%.
- Use of CASE tools leads to considerable improvements to quality. This is mainly due to the facts that one can effortlessly iterate through the different phases of software development and the chances of human error are considerably reduced.
- CASE tools help produce high quality and consistent documents. Since the important data relating to a software product are maintained in a central

repository, redundancy in the stored data is reduced and therefore chances of inconsistent documentation is reduced to a great extent.

- CASE tools take out most of the drudgery in a software engineer's work. For example, they need not check meticulously the balancing of the DFDs but can do it effortlessly through the press of a button.
- CASE tools have led to revolutionary cost saving in software maintenance efforts. This arises not only due to the tremendous value of a CASE environment in traceability and consistency checks, but also due to the systematic information capture during the various phases of software development as a result of adhering to a CASE environment.
- Introduction of a CASE environment has an impact on the style of working of a company, and makes it oriented towards the structured and orderly approach.

15.1.2 Requirements of a prototyping CASE tool

Prototyping is useful to understand the requirements of complex software products, to demonstrate a concept, to market new ideas, and so on. The important features of a prototyping CASE tool are as follows:

- Define user interaction
- Define the system control flow
- Store and retrieve data required by the system
- Incorporate some processing logic

15.1.2.1 Features of a good prototyping CASE tool

There are several stand-alone prototyping tools. But a tool that integrates with the data dictionary can make use of the entries in the data dictionary, help in populating the data dictionary and ensure the consistency between the design data and the prototype. A good prototyping tool should support the following features:

- Since one of the main uses of a prototyping CASE tool is graphical user interface (GUI) development, prototyping CASE tool should support the user to create a GUI using a graphics editor. The user should be allowed to define all data entry forms, menus and controls.
- It should integrate with the data dictionary of a CASE environment.

- If possible, it should be able to integrate with external user defined modules written in C or some popular high-level programming languages.
- The user should be able to define the sequence of states through which a created prototype can run. The user should also be allowed to control the running of the prototype.
- The run time system of prototype should support mock runs of the actual system and management of the input and output data.

15.1.2.2 Structured analysis and design with CASE tools

Several diagramming techniques are used for structured analysis and structured design. The following supports might be available from CASE tools.

- A CASE tool should support one or more of the structured analysis and design techniques.
- It should support effortlessly drawing analysis and design diagrams.
- It should support drawing for fairly complex diagrams, preferably through a hierarchy of levels.
- The CASE tool should provide easy navigation through the different levels and through the design and analysis.
- The tool must support completeness and consistency checking across the design and analysis and through all levels of analysis hierarchy. Whenever it is possible, the system should disallow any inconsistent operation, but it may be very difficult to implement such a feature. Whenever there arises heavy computational load while consistency checking, it should be possible to temporarily disable consistency checking.

15.1.2.3 Code generation and CASE tools

As far as code generation is concerned, the general expectation of a CASE tool is quite low. A reasonable requirement is traceability from source file to design data. More pragmatic supports expected from a CASE tool during code generation phase are the following:

- The CASE tool should support generation of module skeletons or templates in one or more popular languages. It should be possible to

include copyright message, brief description of the module, author name and the date of creation in some selectable format.

- The tool should generate records, structures, class definition automatically from the contents of the data dictionary in one or more popular languages.
- It should generate database tables for relational database management systems.
- The tool should generate code for user interface from prototype definition for X window and MS window-based applications.

15.1.2.4 Test case generation CASE tool

The CASE tool for test case generation should have the following features:

- It should support both design and requirement testing.
- It should generate test set reports in ASCII format which can be directly imported into the test plan document.

Check Your Progress I

Mark all options which are true.

1. Computer Aided Software Engineering (CASE) tools can assist in
 - a. phase related activities such as specification, structured analysis, coding, testing, etc.
 - b. non-phase related activities such as software project management, software configuration management, etc.
 - c. neither phase related activities nor non-phase related activities
2. The primary objective(s) in using any CASE tool is(are):
 - a. to increase productivity of software development
 - b. to decrease software development as well as software maintenance cost
 - c. to help produce better quality software
 - d. all of the above

Mark the following statements as either True or False.

3. A programming environment is an integrated collection of tools supporting the activities related to all the phases of software development life cycle.

15.2 DIFFERENT CHARACTERISTICS OF CASE TOOLS

15.2.1 Hardware and environmental requirements

In most cases, it is the existing hardware that would place constraints upon the CASE tool selection. Thus, instead of defining hardware requirements for a CASE tool, the task at hand becomes to fit in an optimal configuration of CASE tool in the existing hardware capabilities. Therefore, it can be emphasized on selecting the most optimal CASE tool configuration for a given hardware configuration.

The heterogeneous network is one instance of distributed environment and this can be chosen for illustration as it is more popular due to its machine independent features. The CASE tool implementation in heterogeneous network makes use of client-server paradigm. The multiple clients who run different modules access data dictionary through this server. The data dictionary server may support one or more projects. Though it is possible to run many servers for different projects but distributed implementation of data dictionary is not common.

The tool set is integrated through the data dictionary which supports multiple projects, multiple users working simultaneously and allows to share information between users and projects. The data dictionary provides consistent view of all project entities, e.g. a data record definition and its entity-relationship diagram be consistent. The server should depict the per-project logical view of the data dictionary. This means that it should allow back up/restore, copy, cleaning part of the data dictionary, etc.

The tool should work satisfactorily for maximum possible number of users working simultaneously. The tool should support multi-windowing environment for the users. This is important to enable the users to see more than one diagram at a time. It also facilitates navigation and switching from one part to the other.

Documentation support

The deliverable documents should be organized graphically and should be able to incorporate text and diagrams from the central repository. This helps in producing up-to-date documentation. The CASE tool should integrate with one or more of the commercially available desktop publishing packages. It should be possible to

export text, graphics, tables, data dictionary reports to the DTP package in standard forms such as PostScript.

Project management support

The CASE tool should support collecting, storing, and analyzing information on the software project's progress such as the estimated task duration, scheduled and actual task start, completion date, dates and results of the reviews, etc.

External interface

The CASE tool should allow exchange of information for reusability of design. The information which is to be exported by the CASE tool should be preferably in ASCII format and support open architecture. Similarly, the data dictionary should provide a programming interface to access information. It is required for integration of custom utilities, building new techniques, or populating the data dictionary.

Reverse engineering

The CASE tool should support generation of structure charts and data dictionaries from the existing source codes. It should populate the data dictionary from the source code. If the tool is used for re-engineering information systems, it should contain conversion tool from indexed sequential file structure, hierarchical and network database to relational database systems.

Data dictionary interface

The data dictionary interface should provide view and update access to the entities and relations stored in it. It should have print facility to obtain hard copy of the viewed screens. It should provide analysis reports like cross-referencing, impact analysis, etc. Ideally, it should support a query language to view its contents.

Second-generation CASE tool

An important feature of the second-generation CASE tool is the direct support of any adapted methodology. This would necessitate the function of a CASE administrator organization who can tailor the CASE tool to a particular methodology. In addition, the second-generation CASE tools have following features:

- **Intelligent diagramming support.** The fact that diagramming techniques are useful for system analysis and design is well established. The future

CASE tools would provide help to aesthetically and automatically lay out the diagrams.

- **Integration with implementation environment.** The CASE tools should provide integration between design and implementation.
- **Data dictionary standards.** The user should be allowed to integrate many development tools into one environment. It is highly unlikely that any one vendor will be able to deliver a total solution. Moreover, a preferred tool would require tuning up for a particular system. Thus the user would act as a system integrator. This is possibly only if some standard on data dictionary emerges.
- **Customization support.** The user should be allowed to define new types of objects and connections. This facility may be used to build some special methodologies. Ideally it should be possible to specify the rules of a methodology to a rule engine for carrying out the necessary consistency checks.

15.2.2 Architecture of a CASE environment

The architecture of a typical modern CASE environment is shown diagrammatically in fig. 15.2. The important components of a modern CASE environment are user interface, tool set, object management system (OMS), and a repository. Characteristics of a tool set have been discussed earlier.

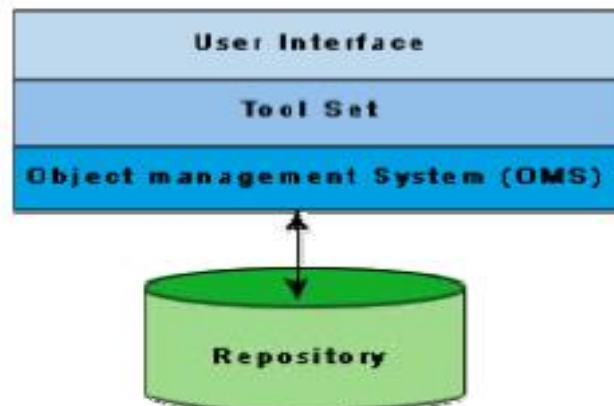


Fig. 15.2: Architecture of a Modern CASE Environment

15.2.2.1 User Interface

The user interface provides a consistent framework for accessing the different tools thus making it easier for the users to interact with the different tools and reducing the overhead of learning how the different tools are used.

15.2.2.2 Object Management System (OMS) and Repository

Different case tools represent the software product as a set of entities such as specification, design, text data, project plan, etc. The object management system maps these logical entities such into the underlying storage management system (repository). The commercial relational database management systems are geared towards supporting large volumes of information structured as simple relatively short records. There are a few types of entities but large number of instances. By contrast, CASE tools create a large number of entity and relation types with perhaps a few instances of each. Thus, the object management system takes care of appropriately mapping into the underlying storage management system.

Check Your Progress II

Mark all options which are true.

1. Which of the following features are related to a prototyping CASE tool?
 - a. to define user interaction
 - b. to define the control flow of the system
 - c. to incorporate some processing logic
 - d. all of the above
2. Which of the following supports should we expect from a CASE tool during the code generation phase of a software development project?
 - a. generation of module skeletons or templates in one or more popular languages
 - b. generation of records, structures, class definition automatically from the contents of the data dictionary in one or more popular languages
 - c. generation of database tables for relational database management systems
 - d. all of the above
3. Which of the following features that are not present in current CASE tools but are likely to be supported as a second-generation CASE tool?

- a. diagramming support
- b. documentation support
- c. customization support for different development methodologies
- d. querying data dictionary

Mark the following statements as either True or False.

4. Use of CASE tools typically leads to improvements to the quality of a software product.
5. CASE tools help in producing c.

15.3 MODEL QUESTIONS

1. What do you understand by the term CASE tool?
2. What are the primary objectives of a CASE tool?
3. What do you understand by the term CASE environment?
4. Differentiate between the characteristics of a CASE environment and a programming environment.
5. What are the main advantages of using CASE tools?
6. Discuss the role of the data dictionary in a CASE environment.
7. What features are supported by a good prototyping CASE tool?
8. Discuss the supports available from CASE tools in order to perform structured analysis and software design activity.
9. During code generation what supports might be expected from CASE tools?
10. How can CASE tool help for the purpose of test case generation?
11. Discuss hardware and environmental requirements for a CASE tool.
12. What are the software project management supports that might be expected from CASE tools?
13. What are the software reverse engineering supports that might be available from CASE tools?
14. What are some of the important features that a future generation CASE tool should support?

15. Schematically draw the architecture of a CASE environment and explain how the different tools are integrated.

UNIT XVI: SOFTWARE REUSE

16.0 LEARNING OBJECTIVES

At the end of this lesson the student would be able to:

- Explain the advantages of software reuse.
- Identify the artifacts that can be reused during software development.
- Explain the pros and cons of knowledge reused.
- Explain why reuse of commonly used mathematical functions is easy to achieve.
- Identify the basic issues that must be clearly addressed for starting any reuse program.
- Explain what is meant by domain analysis.
- Explain a scheme by which software reusable components can be satisfactorily classified.
- Search an item from the domain repository.
- Explain how a reuse repository can be maintained.
- Explain what is meant by an application generator.
- Identify the advantages of using an application generator compared to parameterized programs.
- Identify the shortcomings of application generator.
- Identify the steps that can be adopted for achieving organization-level reuse.
- Identify the non-technical factors that inhibit an effective reuse program.

16.1 BASIC IDEAS ON SOFTWARE REUSE

16.1.1 Advantages of software reuse

Software products are expensive. Software project managers are worried about the high cost of software development and are desperately look for ways to cut development cost. A possible way to reduce development cost is to reuse parts

from previously developed software. In addition to reduced development cost and time, reuse also leads to higher quality of the developed products since the reusable components are ensured to have high quality.

16.1.2 Artifacts that can be reused

It is important to know about the kinds of the artifacts associated with software development that can be reused. Almost all artifacts associated with software development, including project plan and test plan can be reused. However, the prominent items that can be effectively reused are:

- Requirements specification
- Design
- Code
- Test cases
- Knowledge

16.1.3 Pros and cons of knowledge reuse

Knowledge is the most abstract development artifact that can be reused. Out of all the reuse artifacts i.e. requirements specification, design, code, test cases, reuse of knowledge occurs automatically without any conscious effort in this direction. However, two major difficulties with unplanned reuse of knowledge are that a developer experienced in one type of software product might be included in a team developing a different type of software. Also, it is difficult to remember the details of the potentially reusable development knowledge. A planned reuse of knowledge can increase the effectiveness of reuse. For this, the reusable knowledge should be systematically extracted and documented. But, it is usually very difficult to extract and document reusable knowledge.

16.1.4 Easiness of reuse of mathematical functions

The routines of mathematical libraries are being reused very successfully by almost every programmer. No one in his right mind would think of writing a routine to compute sine or cosine. Reuse of commonly used mathematical functions is easy. Several interesting aspects emerge. Cosine means the same to all. Everyone has clear ideas about what kind of argument should cosine take, the

type of processing to be carried out and the results returned. Secondly, mathematical libraries have a small interface. For example, cosine requires only one parameter. Also, the data formats of the parameters are standardized.

16.1.5 Basic issues in any reuse program

The following are some of the basic issues that must be clearly understood for starting any reuse program.

- Component creation
- Component indexing and storing
- Component search
- Component understanding
- Component adaptation
- Repository maintenance

Component creation. For component creation, the reusable components have to be first identified. Selection of the right kind of components having potential for reuse is important. Domain analysis is a promising technique which can be used to create reusable components.

Component indexing and storing. Indexing requires classification of the reusable components so that they can be easily searched when looking for a component for reuse. The components need to be stored in a Relational Database Management System (RDBMS) or an Object-Oriented Database System (ODBMS) for efficient access when the number of components becomes large.

Component searching. The programmers need to search for right components matching their requirements in a database of components. To be able to search components efficiently, the programmers require a proper method to describe the components that they are looking for.

Component understanding. The programmers need a precise and sufficiently complete understanding of what the component does to be able to decide whether they can reuse the component. To facilitate understanding, the components should be well documented and should do something simple.

Component adaptation. Often, the components may need adaptation before they can be reused, since a selected component may not exactly fit the problem at hand.

However, tinkering with the code is also not a satisfactory solution because this is very likely to be a source of bugs.

Repository maintenance. A component repository once is created requires continuous maintenance. New components, as and when created have to be entered into the repository. The faulty components have to be tracked. Further, when new applications emerge, the older applications become obsolete. In this case, the obsolete components might have to be removed from the repository.

16.1.6 Domain analysis

The aim of domain analysis is to identify the reusable components for a problem domain.

Reuse domain. A reuse domain is a technically related set of application areas. A body of information is considered to be a problem domain for reuse, if a deep and comprehensive relationship exists among the information items as categorized by patterns of similarity among the development components of the software product. A reuse domain is shared understanding of some community, characterized by concepts, techniques, and terminologies that show some coherence. Examples of domains are accounting software domain, banking software domain, business software domain, manufacturing automation software domain, telecommunication software domain, etc.

Just to become familiar with the vocabulary of a domain requires months of interaction with the experts. Often, one needs to be familiar with a network of related domains for successfully carrying out domain analysis. Domain analysis identifies the objects, operations, and the relationships among them. For example, consider the airline reservation system, the reusable objects can be seats, flights, airports, crew, meal orders, etc. The reusable operations can be scheduling a flight, reserving a seat, assigning crew to flights, etc. The domain analysis generalizes the application domain. A domain model transcends specific applications. The common characteristics or the similarities between systems are generalized.

During domain analysis, a specific community of software developers gets together to discuss community-wide-solutions. Analysis of the application domain is required to identify the reusable components. The actual construction of reusable components for a domain is called domain engineering.

Evolution of a reuse domain. The ultimate result of domain analysis is development of problem-oriented languages. The problem-oriented languages are also known as application generators. These application generators, once developed form application development standards. The domains slowly develop. As a domain develops, it is distinguishable the various stages it undergoes:

Stage 1: There is no clear and consistent set of notations. Obviously, no reusable components are available. All software is written from scratch.

Stage 2: Here, only experience from similar projects is used in a development effort. This means that there is only knowledge reuse.

Stage 3: At this stage, the domain is ripe for reuse. The set of concepts are stabilized and the notations standardized. Standard solutions to standard problems are available. There is both knowledge and component reuse.

Stage 4: The domain has been fully explored. The software development for the domain can be largely automated. Programs are not written in the traditional sense any more. Programs are written using a domain specific language, which is also known as an application generator.

Check Your Progress I

Mark all options which are true.

1. Component-based software development leads to
 - a. high quality software product
 - b. reduced development cost
 - c. reduced development time
 - d. all of the above
2. Which of the following kinds of artifacts associated with software development can be reused?
 - a. requirements specification
 - b. design
 - c. code
 - d. knowledge
 - e. all of the above

3. The most abstract artifact associated with software development that can be reused is
 - a. requirements specification
 - b. design
 - c. code
 - d. knowledge
 - e. test cases
4. For efficient access, the reusable components are needed to be stored in which of the following systems?
 - a. Relational Database Management System (RDBMS)
 - b. Object-Oriented Database System (ODBMS)
 - c. both of RDBMS and ODBMS
5. Domain analysis identifies which of the following?
 - a. objects
 - b. operations
 - c. relationships among objects
 - d. all of the above

Mark the following statements as either True or False.

6. We can easily create components that can be reused in different software development applications.
7. The reuse of commonly used mathematical functions is very much easy.
8. Classification of reusable components is very much required for component indexing and storage.

16.2 REUSE APPROACH

16.2.1 Components classification

Components need to be properly classified in order to develop an effective indexing and storage scheme. Hardware reuse has been very successful. Hardware components are classified using a multilevel hierarchy. At the lowest level, the components are described in several forms: natural language description, logic schema, timing information, etc. The higher the level at which a component is

described, the more is the ambiguity. This has motivated the Prieto-Diaz's classification scheme.

Prieto-Diaz's classification scheme: Each component is best described using a number of different characteristics or facets. For example, objects can be classified using the following:

- actions they embody
- objects they manipulate
- data structures used
- systems they are part of, etc.

Prieto-Diaz's faceted classification scheme requires choosing an n-tuple that best fits a component. Faceted classification has advantages over enumerative classification. Strictly enumerative schemes use a predefined hierarchy. Therefore, these forces to search for an item that best fit the component to be classified. This makes it very difficult to search a required component. Though cross-referencing to other items can be included, the resulting network becomes complicated.

16.2.2 Searching

The domain repository may contain thousands of reuse items. A popular search technique that has proved to be very effective is one that provides a web interface to the repository. Using such a web interface, one would search an item using an approximate automated search using key words, and then from these results do a browsing using the links provided to look up related items. The approximate automated search locates products that appear to fulfill some of the specified requirements. The items located through the approximate search serve as a starting point for browsing the repository. These serve as the starting point for browsing the repository. The developer may follow links to other products until a sufficiently good match is found. Browsing is done using the keyword-to-keyword, keyword-to-product, and product-to-product links. These links help to locate additional products and compare their detailed attributes. Finding a satisfactorily item from the repository may require several locations of approximate search followed by browsing. With each iteration, the developer would get a better understanding of the available products and their differences.

However, we must remember that the items to be searched may be components, designs, models, requirements, and even knowledge.

16.2.3 Repository maintenance

Repository maintenance involves entering new items, retiring those items which are no more necessary, and modifying the search attributes of items to improve the effectiveness of search. The software industry is always trying to implement something that has not been quite done before. As patterns requirements emerge, new reusable components are identified, which may ultimately become more or less the standards. However, as technology advances, some components which are still reusable, do not fully address the current requirements. On the other hand, restricting reuse to highly mature components, sacrifices one of the creates potential reuse opportunity. Making a product available before it has been thoroughly assessed can be counter productive. Negative experiences tend to dissolve the trust in the entire reuse framework.

16.2.4 Application generator

The problem-oriented languages are known as application generators. Application generators translate specifications into application programs. The specification is usually written using 4GL. The specification might also in a visual form. Application generator can be applied successfully to data processing application, user interface, and compiler development.

16.2.4.1 Advantages of application generators

Application generators have significant advantages over simple parameterized programs. The biggest of these is that the application generators can express the variant information in an appropriate language rather than being restricted to function parameters, named constants, or tables. The other advantages include fewer errors, easier to maintain, substantially reduced development effort, and the fact that one need not bother about the implementation details.

16.2.4.2 Shortcomings of application generator

Application generators are handicapped when it is necessary to support some new concepts or features. Application generators are less successful with the

development of applications with close interaction with hardware such as real-time systems.

16.2.5 Re-use at organization level

Achieving organization-level reuse requires adoption of the following steps:

- Assessing a product's potential for reuse
- Refining products for greater reusability
- Entering the product in the reuse repository

Assessing a product's potential for reuse. Assessment of components reuse potential can be obtained from an analysis of a questionnaire circulated among the developers. The questionnaire can be devised to assess a component's reusability. The programmers working in similar application domain can be used to answer the questionnaire about the product's reusability. Depending on the answers given by the programmers, either the component be taken up for reuse as it is, it is modified and refined before it is entered into the reuse repository, or it is ignored. A sample questionnaire to assess a component's reusability is the following.

- Is the component's functionality required for implementation of systems in the future?
- How common is the component's function within its domain?
- Would there be a duplication of functions within the domain if the component is taken up?
- Is the component hardware dependent?
- Is the design of the component optimized enough?
- If the component is non-reusable, then can it be decomposed to yield some reusable components?
- Can we parameterize a non-reusable component so that it becomes reusable?

Refining products for greater reusability. For a product to be reusable, it must be relatively easy to adapt it to different contexts. Machine dependency must be abstracted out or localized using data encapsulation techniques. The following refinements may be carried out:

- **Name generalization:** The names should be general, rather than being directly related to a specific application.
- **Operation generalization:** Operations should be added to make the component more general. Also, operations that are too specific to an application can be removed.
- **Exception generalization:** This involves checking each component to see which exceptions it might generate. For a general component, several types of exceptions might have to be handled.
- **Handling portability problems:** Programs typically make some assumption regarding the representation of information in the underlying machine. These assumptions are in general not true for all machines. The programs also often need to call some operating system functionality and these calls may not be same on all machines. Also, programs use some function libraries, which may not be available on all host machines. A portability solution to overcome these problems is shown in fig. 16.1. The portability solution suggests that rather than call the operating system and I/O procedures directly, abstract versions of these should be called by the application program. Also, all platform-related calls should be routed through the portability interface. One problem with this solution is the significant overhead incurred, which makes it inapplicable to many real-time systems and applications requiring very fast response.

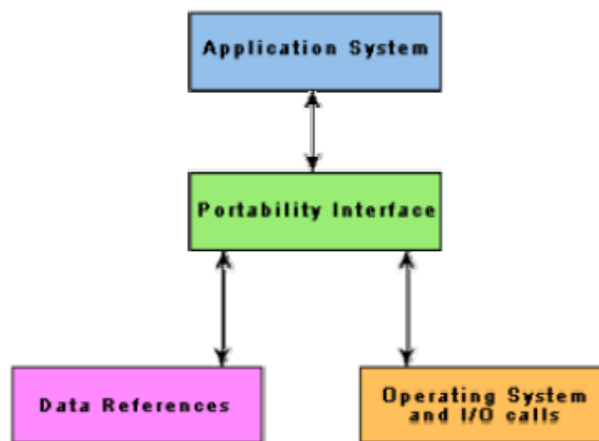


Fig. 16.1: Improving reusability of a component by using a portability interface

16.2.6 Factors that inhibit an effective reuse program

In spite of all the shortcomings of the state-of-the-art reuse techniques, it is the experience of several organizations that most of the factors inhibiting an effective reuse program are non-technical. Some of these factors are the following.

- Need for commitment from the top management.
- Adequate documentation to support reuse.
- Adequate incentive to reward those who reuse. Both the people contributing new reusable components and those reusing the existing components should be rewarded to start a reuse program and keep it going.
- Providing access to and information about reusable components. Organizations are often hesitant to provide an open access to the reuse repository for the fear of the reuse components finding a way to their competitors.

Check Your Progress II

Mark all options which are true.

1. The actual construction of the reusable components for a domain is called
 - a. domain analysis
 - b. domain engineering
 - c. component creation
 - d. none of the above
2. Application generators can successfully be applied to
 - a. data processing application
 - b. user interface
 - c. compiler development
 - d. all of the above
3. Which of the following steps is required to achieve successful organization-level reuse?
 - a. assess of an item's potential for reuse
 - b. refine the item for greater reusability
 - c. enter the product in the reuse repository
 - d. all of the above

Mark the following statements as either True or False.

4. A component repository requires continuous maintenance.
5. Application generators translate specifications into application programs.

16.3 MODEL QUESTIONS

1. Why is it important for an organization to undertake an effective reuse program?
2. What are the important artifacts that can be reused?
3. Why is reuse of software components much more difficult than hardware components?
4. Do you agree with the statement: “code” is the most important reuse artifact that can be used during software development.
5. Identify the reasons why reuse of mathematical software is so successful. Also, identify the reasons why the reuse of software components other than those of the mathematical software is difficult.
6. What are the issues that must be clearly understood for starting any reuse program?
7. Devise a scheme to store software reuse artifacts. Explain how components can be searched in that scheme.
8. What do you understand by the term reuse domain?
9. How does domain analysis increase software reusability?
10. Identify the stages through which a reuse domain progresses.
11. What do you understand by the term “faceted classification” in the context of software reuse? How does faceted classification simplify component search in a component store?
12. What is meant by the term “application generator”?
13. Why reuse is easier while using an application generator compared to a component library?
14. What are the shortcomings of an application generator?
15. How can you improve reusability of the components you have identified for reuse during program development?

UNIT XVII: CLIENT-SERVER SOFTWARE DEVELOPMENT

17.0 LEARNING OBJECTIVES

At the end of this lesson the student would be able to:

- Explain what is a client-server software.
- Explain the advantages of client-server software over centralized solutions.
- Explain the factors responsible for making client-server solutions feasible, affordable, and popular in recent times.
- Identify the advantages of client-server software development compared to monolithic ones.
- Identify the disadvantages of client-server software.
- Differentiate between host-slave computing and client-server computing. Give an example for each.
- Explain the two-tier client-server architecture.
- Explain the limitations of two-tier client-server architecture.
- Explain three-tier client-server architecture.
- Identify the functions of middleware.
- Identify the popular middleware standards.
- Explain what Common Object Request Broker Architecture (CORBA) is.
- Explain CORBA reference model.
- Explain CORBA architecture.
- Identify the functions of Object Request Broker (ORB).
- Identify the commercial ORBs.
- Explain what is stub.
- Explain Dynamic Invocation Interface (DII) in CORBA.
- Explain what is Component Object Model (COM).
- Explain what is Distributed Component Object Model (DCOM).
- Explain Inter-ORB communication.
- Identify the features of General Inter-ORB Protocol (GIOP).

- Differentiate between CORBA based development and COM/DCOM development.

17.1 BASIC IDEAS ON CLIENT-SERVER SOFTWARE DEVELOPMENT AND CLIENT-SERVER ARCHITECTURE

17.1.1 Client-server software

A client is basically a consumer of services and server is a provider of services as shown in fig. 17.1. A client requests some services from the server and the server provides the required services to the client. Client and server are usually software components running on independent machines. Even a single machine can sometimes acts as a client and at other times a server depending on the situations. Thus, client and server are mere roles.

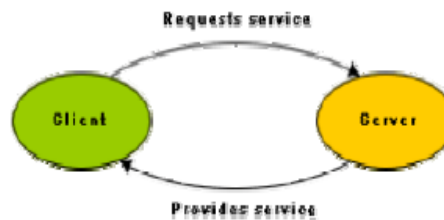


Fig. 17.1: Client-server model

Example:

A man was visiting his friend's town in his car. The man had a handheld computer (client). He knew his friend's name but he didn't know his friend's address. So he sent a wireless message (request) to the nearest "address server" by his handheld computer to enquire his friend's address. The message first came to the base station. The base station forwarded that message through landline to local area network where the server is located. After some processing, LAN sent back that friend's address (service) to the man.

17.1.1.1 Advantages of client-server software

The client-server software architecture is a versatile, message-based and modular infrastructure that is intended to improve usability, flexibility, interoperability and scalability as compared to centralized, mainframe, time sharing computing.

17.1.1.2 Factors for feasibility and popularity of client-server solutions

Client-server concept is not a new concept. It already existed in the society for long time. A doctor is a client of a barber, who in turn is a client of the lawyer and so forth. Something can be a server in some context and a client in some other context. So client and server are mere roles as shown in fig. 17.2.

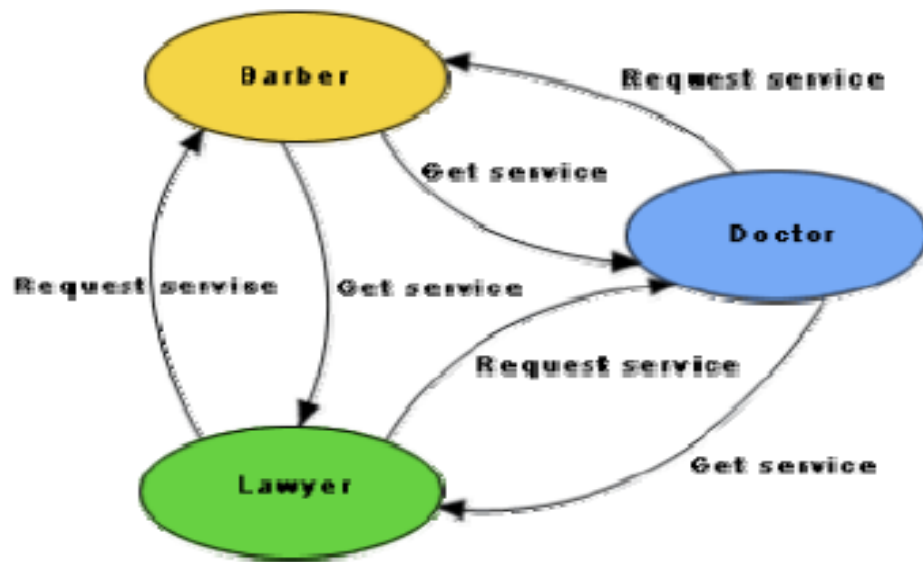


Fig. 17.2: Client and server as roles

There are many reasons for the popularity of client-server software development. Some reasons are:

- Computers have become small, decentralized and cheap
- Networking has become affordable, reliable, and efficient.
- Client-server systems divide up the work of computing among many separate machines. Thus, client-server solutions are modular and loosely coupled. So they are easy to develop and maintain.

17.1.1.3 Advantages of client-server software development

There are many advantages of client-server software products as compared to monolithic ones. These advantages are:

- **Simplicity and modularity** – Client and server components are loosely coupled and therefore modular. These are easy to understand and develop.

- **Flexibility** – Both client and server software can be easily migrated across different machines in case some machine becomes unavailable or crashes. The client can access the service anywhere. Also, clients and servers can be added incrementally.
- **Extensibility** – More servers and clients can be effortlessly added.
- **Concurrency** – The processing is naturally divided across several machines. Clients and servers reside in different machines which can operate in parallel and thus processing becomes faster.
- **Cost Effectiveness** – Clients can be cheap desktop computers whereas servers can be sophisticated and expensive computers. To use a sophisticated software, one needs to own only a cheap client and invoke the server.
- **Specialization** – One can have different types of computers to run different types of servers. Thus, servers can be specialized to solve some specific problems.
- **Current trend** – Mobile computing implicitly uses client-server technique. Cell phones (handheld computers) are being provided with small processing power, keyboard, small memory, and LCD display. Cell phones cannot really compute much as they have very limited processing power and storage capacity but they can act as clients. The handheld computers only support the interface to place requests on some remote servers.
- **Application Service Providers (ASPs)** – There are many application software products which are very expensive. Thus, it makes prohibitively costly to own those applications. The cost of those applications often runs into millions of dollars. For example, a Chemical Simulation Software named “Aspen” is very expensive but very powerful. For small industries it would not be practical to own that software. Application Service Providers can own ASPEN and let the small industries use it as client and charge them based on usage time. A client can simply log in and ASP charges according to the time that the software is used.

- **Component-based development** – It is the enabler of the client-server technology. Component-based development is radically different from traditional software development. In component-based development, a developer essentially integrates pre-built components purchased off-the-shelf. This is akin to the way hardware developers integrate ICs on a Printed Circuit Board (PCB). Components might reside on different computers which act as servers and clients.
- **Fault-tolerance** – Client-server based systems are usually fault-tolerant. There can be many servers. If one server crashes then client requests can be switched to a redundant server.

There are many other advantages of client-server software. For example, we can locate a server near to the client. There might be several servers and the client requests can be routed to the nearest server. This would reduce the communication overhead.

17.1.1.4 Disadvantages of client-server software

There are several disadvantages of client-server software development. Those disadvantages are:

- **Security** – In a monolithic application, implementation of security is very easy. But in a client-server based development a lot of flexibility is provided and a client can connect from anywhere. This makes it easy for hackers to break into the system. Therefore, ensuring security in client-server system is very challenging.
- **Servers can be bottlenecks** – Servers can turn out to be bottlenecks because many clients might try to connect to a server at the same time. This problem arises due to the flexibility given that any client can connect anytime required.
- **Compatibility** – Clients and servers may not be compatible to each other. Since the client and server components may be manufactured by different vendors, they may not be compatible with respect to data types, language, etc.
- **Inconsistency** – Replication of servers is a problem as it can make data inconsistent.

17.1.1.5 Host-slave computing vs. client-server computing

An example of a host-slave computing is a Railway-reservation system. The software is divided into two parts – one resides on the terminals of the booking clerks. The master at any time directs the slaves what to do. A slave can only make requests and master takes over and tells what to do.

On the other hand, in a client-server computing, different components are interfaced using an open protocol. In a master-slave they are proprietary. An example of a client-server system is a world wide web.

17.1.2 Two-tier client-server architecture

The simplest way to connect clients and servers is a two-tier architecture as shown in fig. 17.3. In a two-tier architecture, any client can get service from any server by initiating a request over the network. With two tier client-server architectures, the user interface is usually located in the user's desktop and the services are usually supported by a server that is a powerful machine that can service many clients. Processing is split between the user interface and the database management server. There are a number of software vendors who provide tools to simplify development of applications for the two-tier client-server architecture.

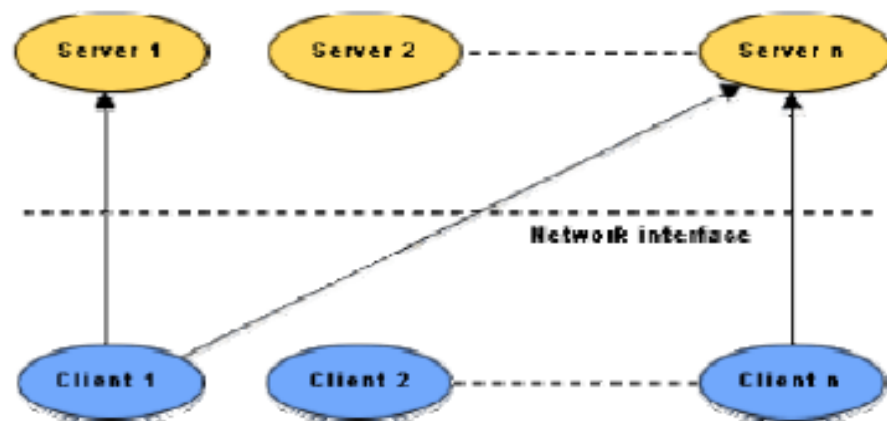


Fig. 17.3: Two-tier client-server architecture

17.1.2.1 Limitations of two-tier client-server architecture

A two-tier architecture for client-server applications is an ideal solution but is not practical. The problem is that client and server components are manufactured by

different vendors and the different vendors come up with different sets of interfaces and different implementation standards. That's the reason why clients and servers can often not talk to each other. A two-tier architecture can work only in an open environment. In an open environment all components have standard interfaces. However, till date an open environment is still far from becoming practical.

17.1.3 Three-tier client-server architecture

The three-tier architecture overcomes the important limitations of the two-tier architecture. In the three-tier architecture, a middleware was added between the user system interface client environment and the server environment as shown in fig. 17.4. The middleware keeps track of all server locations. It also translates client's requests into server understandable form. For example, if the middleware provides queuing, the client can deliver its request to the middleware and disengage because the middleware will access the data and return the answer to the client.

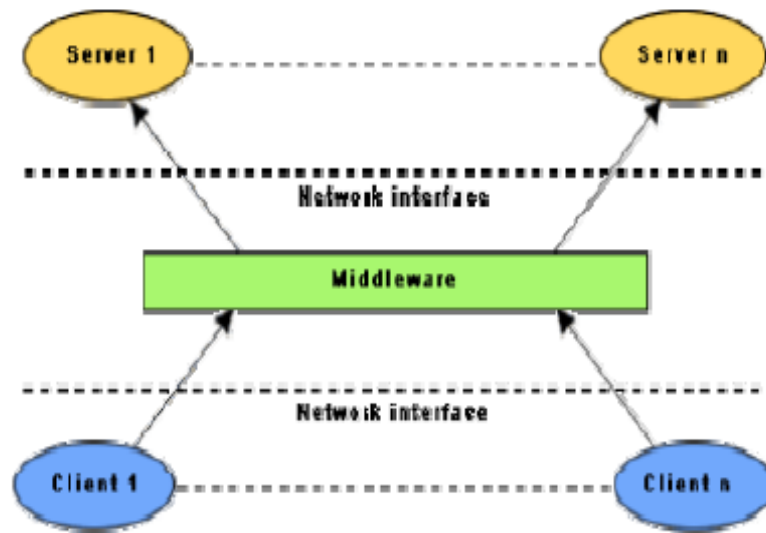


Fig. 17.4: Three-tier client-server architecture

17.1.3.1 Functions of middleware

The middleware performs many activities such as:

- It knows the addresses of servers. So, based on client requests, it can locate the servers.
- It can translate between client and server formats of data and vice versa.

17.1.3.2 Popular middleware standards

Two popular middleware standards are:

- CORBA (Common Object Request Broker Architecture)
- COM/DCOM

CORBA is being promoted by Object Management Group (OMG), a consortium of a large number of computer industries such as IBM, HP, Digital etc. Actually OMG is not a standards body, they only try to promote de facto standards. They don't have any authority to make or enforce standards. They just try to popularize good solutions with the hope that if they become highly popular, they would automatically become standard. COM/DCOM is being promoted by Microsoft alone.

Check Your Progress I

Mark all options which are true.

1. What are the reasons for the recent popularity of the client-server style of software development?
 - a. computers have become small, decentralized and cheap
 - b. networking has become affordable, reliable, and efficient
 - c. client-server systems divide up the work of computing among many separate machines
 - d. all of the above
2. Which of the following functions are performed by middleware?
 - a. it can identify the server from either its id or its service type
 - b. it knows client protocols and server protocols
 - c. it can deliver client-request to the server and server-response to the client
 - d. all of the above

3. Which of the following domain independent services are provided by object interface in an Object Management Architecture (OMA)?
 - a. naming service
 - b. trading service
 - c. security service
 - d. all of the above
4. Which of the following functions does Object Request Broker (ORB) perform?
 - a. to transmit the client request to the server and get the response back to the client
 - b. location and possible activation of remote objects
 - c. interface definition
 - d. all of the above

Mark the following statements as either True or False.

5. Client-server software development is synonymous with component-based development.
6. Fault-tolerance is more difficult to provide in a monolithic application compared to its client-server implementation.
7. Client-server based software development is more secure than a monolithic software.
8. Two-tier client-server architecture is a practical solution for distributed computing applications.

17.2 CORBA AND COM/DCOM

17.2.1 Common Object Request Broker Architecture (CORBA)

The Common Object Request Broker Architecture (CORBA) is a specification of a standard architecture for middleware.

Using a CORBA implementation, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The middleware takes the call, and is responsible for finding an object that can implement the request, passing it the parameters, invoking its method, and

returning the results of the invocation. The client does not have to be aware of where the object is located, its programming language, its operating system or any other aspects that are not part of an object's interface.

17.2.1.1 CORBA reference model

The CORBA reference model called Object Management Architecture (OMA) is shown in fig. 17.5. The OMA is itself a specification (actually, a collection of related specifications) that defines a broad range of services for building distributed client-server applications. Many services one might expect to find in a middleware product such as CORBA (e.g., naming, transaction, and asynchronous event management services) are actually specified as services in the OMA.

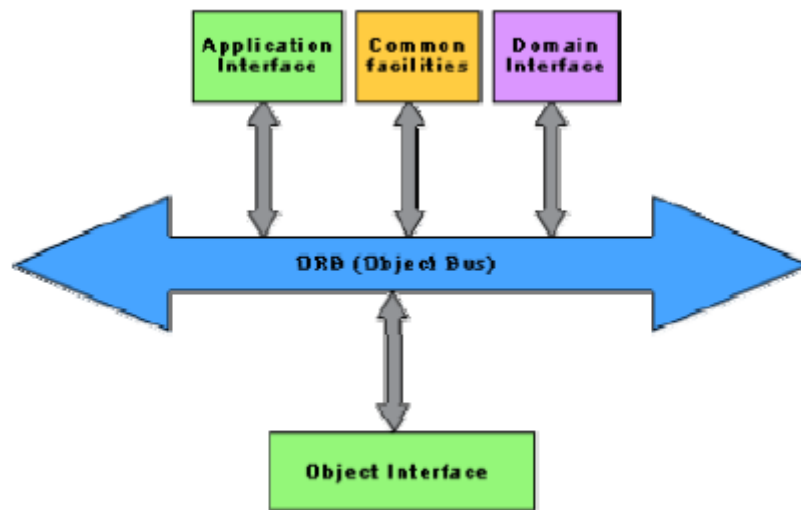


Fig. 17.5: Object Management Architecture (OMA)

- Different components communicate using ORB. ORB is also known as the object bus. An example of application interface is distributed document facility. In a domain interface, it can have domain dependent services, for example, manufacturing domain. Object interface has some domain independent services: **Naming Service:** Naming service is also called white page service. Using naming service server-name can be searched and it's location or address found out.
- **Trading Service:** Trading service is also called yellow page service. Using trading service a specific service can be searched. This is akin to searching a service such as automobile repair shop in a yellow page directory.

There can be other services which can be provided by object interfaces such as security services, life-cycle services and so on.

17.2.1.2 Explain CORBA architecture

Fig. 17.6 depicts the basic components and interfaces defined by CORBA.

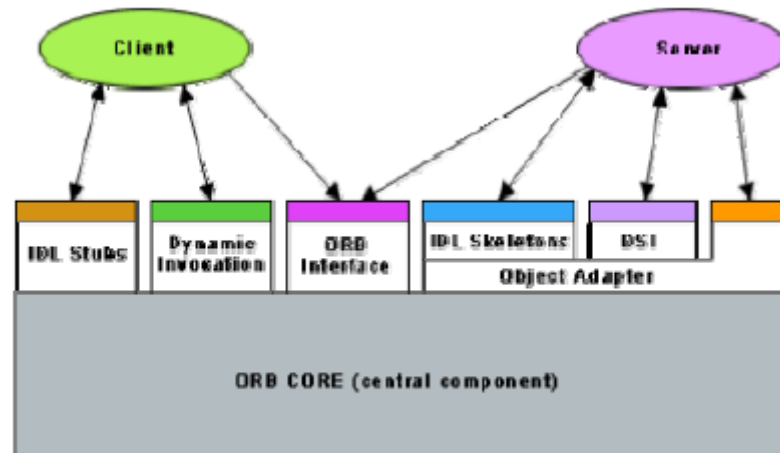


Fig. 17.6: CORBA Architecture

Using a CORBA implementation clients can communicate to the server in two ways:

- Stub
- Dynamic Invocation Interface (DII)

A client can invoke the server services through a Stub and the server gets the requests through the Skeleton. Alternatively, a client can avail services from server through Dynamic Invocation Interface (DII).

17.2.1.3 Functions of Object Request Broker (ORB)

ORB is the central component of the CORBA architecture. The main responsibility of ORB is to transmit the client request to the server and get the response back to the client. ORB abstracts out many procedures involved in service invocation and makes service invocation by client seamless and easy. The main responsibilities of ORB are the following:

- Server location
- Server state management

- Communication between clients and servers

An object request broker provides directory services and helps establish connections between clients and these services [CORBA 96, Steinke 95]. Fig. 17.7 illustrates some of the key ideas.

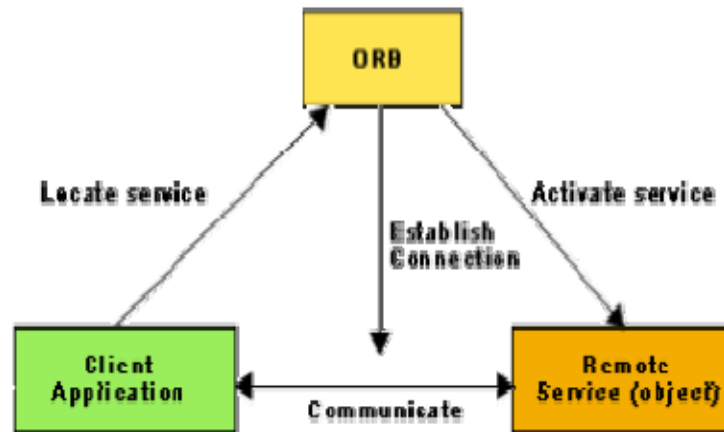


Fig. 17.7: Object Request Broker

The ORB must support a large number of functions in order to operate consistently and effectively. The ORB implements much of these functionality as pluggable modules to simplify the design and implementation of ORB and to make it efficient.

ORB allows objects to hide their implementation details from clients. This can include programming language, operating system, host hardware, and object location.

17.2.1.4 Commercial ORBs

There are several ORBs that are commercially available.

- **Visigenic:** This is probably most popular one. Netscape browser supports Visigenic. CORBA applications can be run using Netscape web browser. In other words, Netscape browser can act as client for CORBA applications. Netscape is extremely popular and there are several millions of copies installed on desktops across the world.
- **IONa**
- **Orbix**
- **Java IDL**

17.2.1.5 Steps to develop application in CORBA

Service can be invoked by a client through either stub or Dynamic Invocation Interface (DII). Before developing a client-server application, the problem is split into two parts: client part and the server part. Next the exact client and server interfaces are determined.

- To specify an interface, IDL (Interface Definition Language) is used.

IDL is very similar to C++ and Java except that it has no executable statements. Using IDL only data interface between clients and servers can be defined. It supports inheritance so that interfaces can be reused. It also supports exception.

After the client-server interface is specified in IDL an IDL compiler is used to compile the IDL specification. Depending on whether the target language in which the application is to be developed is Java, C++, C, etc. IDL2Java, IDL2C++, IDL2C etc. can be used appropriately. When the IDL specification is compiled, it generates the skeletal code for stub and skeleton as shown in fig. 17.8. The stub and skeleton contain interface definition, but the methods (services) are to be filled in by the programmers.

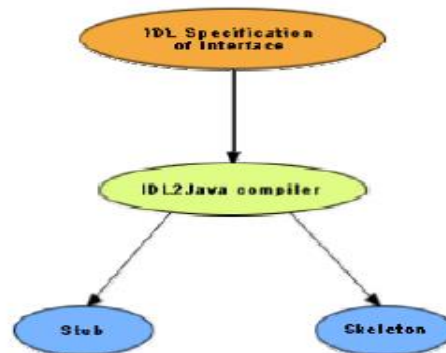


Fig. 17.8: Creation of stub and skeleton using IDL

Service invocation by client through stub is suitable when the interface between the client and server is fixed and it does not change with time. If Interface is known before starting to develop client and the server parts then stubs can effectively be used for service invocation.

The stub part will reside in the client computer and that would basically act as a proxy for the server which may reside in the remote computer. That is the reason why stub is also known as a proxy.

17.2.1.6 Dynamic Invocation Interface (DII) in CORBA

Service invocation through Dynamic Invocation Interface (DII) transparently accesses the interface repository (IR). When an object gets created, it registers information about itself with IR. DII gets the relevant information from the IR and lets the client know about the interface being used. DII is inefficient as compared to stubs.

17.2.2 Component Object Model (COM)

The main idea in the Component Object Model (COM) is that:

- Different vendors can sell binary components.
- Application can be developed by integrating off-the-shelf and proprietary components.

COM runs on a single computer. The concepts used are very similar to CORBA. The components are known as binary objects. These can be generated using languages such as Visual Basic, Delphi, Visual C++ etc. These languages have the necessary features to create COM components. COM components are binary objects and they exist in the form of .exe or .dll (dynamic link library). .exe COM components have separate existence. But .dll COM components are in-process servers. So they get linked to a process. For example, ActiveX is a dll type server. ActiveX can get loaded on the client-side using the dll.

17.2.3 Distributed Component Object Model (DCOM)

Distributed Component Object Model (DCOM) is the extension of the Component Object Model (COM). The restriction that clients and servers reside in the same computer is released here. So, DCOM and CORBA both operate on networked computers. Here development is much easier as compared to CORBA development. Many of the things are transparent to the programmer such as proxy generation, service invocation etc.

17.2.4 Inter-ORB communication

How does ORB do in service invocation when different components exist in different LANS? The answer is Inter-ORB Communication. CORBA 1.0 did not permit Inter-ORB Communication. CORBA 2.0 removes the shortcoming. CORBA 2.0 defines general interoperability standard.

In bridge-based interoperability, the bridge is basically responsible for translating ORB specification information from one ORB to other ORB. For example, one service is known as one reference number in one ORB but that service is known as different reference number in the other ORB.

17.2.5 Features of General Inter-ORB Protocol (GIOP)

The General Inter-ORB Protocol (GIOP) is an abstract meta-protocol. It specifies a standard transfer syntax (how data is represented as bits and bytes) and a set of message formats for object requests. The GIOP is designed to work over many different transport protocols.

The features of GIOP are as follows:

- Designed to be simple, scalable and easy to implement. Every ORB must support GIOP mapped onto local transport.
- GIOP can be used almost any connection-oriented byte stream transport.
- Common Data Representation (CDR) encoding on data types.

One popular connection-oriented transport on which GIOP is implemented is TCP/IP. So the implementation of GIOP on TCP/IP is known as IIOP (Internet Inter-ORB Protocol). It is very popular and frequently used.

17.2.6 CORBA vs. COM/DCOM

If it is the case that all applications reside on PCs and are to run fully on Microsoft platforms then it will be better to use COM/DCOM because development would be much easier here.

- If an application is to be developed for a heterogeneous environment then it will be better to use CORBA.
- Microsoft is very strong on desktop applications i.e. GUI-based applications. Whereas, CORBA-based development is stronger on server side. Java Beans promise to overcome the shortcoming on desktop side i.e. the client part.

Check Your Progress II

Mark all options which are true.

1. Before developing client-server application in CORBA, interface between the client part and the server must specified using
 - a. Interface Definition Language
 - b. Dynamic Invocation Interface
 - c. ORB
 - d. none of the above
2. In CORBA if the server interface would not change with time, then it is more efficient to use
 - a. stubs and skeletons
 - b. DSI and DII
 - c. none of the above
3. In CORBA Dynamic Service Invocation requires
 - a. previous knowledge of the interface between the client and the server part
 - b. Version 2 CSE IIT, Kharagpur
 - c. we do not need to know the interface between the client and the server part
 - d. none of the above
4. What are the properties General Inter-ORB Protocol (GIOP) hold?
 - a. scalable
 - b. easy to implement
 - c. can be used any connection-oriented byte stream transport
 - d. all of the above
5. If some applications run entirely on Microsoft platforms then it will be better to use
 - a. CORBA
 - b. COM/DCOM
 - c. all of the above
6. If clients and servers run on heterogeneous platforms then it will be better to use

- a. CORBA
- b. COM/DCOM
- c. all of the above

Mark the following statements as either True or False.

- 7. The object adapter component in CORBA is responsible for translating the client data formats into server data formats and vice versa.
- 8. The term marshalling in CORBA refers to encryption of client data for added security.
- 9. CORBA is the name of a software product that facilitates development of client-server solutions.
- 10. A CORBA-based client-server solution is constrained to run on a single Local Area Network (LAN).
- 11. Using Internet Inter-ORB Protocol (IIOP), a web browser such as Netscape can serve as a CORBA client.

17.3 MODEL QUESTIONS

- 1. Briefly specify the reasons for the popularity of client-server software development.
- 2. What are the advantages of client-server software development?
- 3. What are the disadvantages of client-server software development?
- 4. Can we say, “Two-tier architecture is the practical solution”? If so, then give the reasons. Briefly specify the limitations of two-tier architecture.
- 5. What are the functions of a middleware in a three-tier architecture? Mention two popular middleware standards.
- 6. Briefly specify the domain independent services provided by object interface in the Object Management Architecture (OMA).
- 7. What are the things that Object Request Broker (ORB) abstracts out? Mention the functions of ORB. Mention some available ORBs.
- 8. How does Dynamic Invocation Interface (DII) know what format data or what exact data required by the server for providing the service and how does the client recognize the data?

9. What is GIOP? What are the features of General Inter-ORB Protocol (GIOP)?
10. Compare between CORBA based development with COM/DCOM development.

REFERENCE

This course on Software Engineering by Prof. Rajib Mall , available at NPTEL repository under Creative Commons Attribution-NonCommercial-ShareAlike - CC-BY-NC-SA, is adopted by Uttarakhand Open University from for Master of Computer Applications program. All copyrights belong to their respective owners.