KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

Coimbatore – 641 021.

(For the candidates admitted from 2015 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

SUBJECT : SOFTWARE ENGINEERING
SEMESTER : VI
SUBJECT CODE : 15CSU601 **CLASS : III B.Sc.CS**

COURSE OBJECTIVE:

The graduates of the software engineering program shall be able to apply proper theoretical, technical, and practical knowledge of software requirements, analysis, design, implementation, verification and validation, and documentation. This course enables the students to resolve conflicting project objectives considering viable tradeoffs within limitations of cost, time, knowledge, existing systems, and organizations.

COURSE OUTCOME:

- Apply their knowledge of mathematics, sciences, and computer science to the modeling, analysis, and measurement of software artifacts.
- Work effectively as leader/member of a development team to deliver quality software artifacts.
- Analyze, specify and document software requirements for a software system.
- Implement a given software design using sound development practices.
- Verify, validate, assess and assure the quality of software artifacts.
- Design, select and apply the most appropriate software engineering process for a given project, plan for a software project, identify its scope and risks, and estimate its cost and time.
- Express and understand the importance of negotiation, effective work habits, leadership, and good communication with stakeholders, in written and oral forms, in a typical software development environment.

.

UNIT-I

Introduction to Software Engineering: The Evolving Role of Software-Software-Software Myths- A Generic View of process: Software Engineering –A Layered Technology- Process Models: Prescriptive Models- Waterfall Model- Incremental process Models. Evolutionary Process Models: Prototyping, The Spiral Model. Specialized process Models

UNIT-II

Building the Analysis Model: Requirements Analysis-Analysis Modeling Approaches-Data Modeling Concepts: Data Objects-Date attributes-Relationships Cardinality and Modality-Flow Oriented Modeling: Creating Data Flow Model-Creating a Control Flow Model-The Control Specification-The Process Specification- Creating a Behavioral Model.

UNIT-III

Design Engineering: Design with the Context of Software Engineering-Design Process and Design Quality-Design Concepts-Creating An Architectural Design: Software Architecture-Data Design-Architectural Design- Assessing Alternative Architectural Designs-Mapping Data Flow into Software Architecture.

UNIT-IV

Performing User Interface Design: The Golden Rules: Place the User in Control-Reduce the User's Memory Load-Make the Interface Consistent- User Interface Analysis and Design: Interface Analysis and Design Models- The Process- Interface Analysis: User Analysis - Task analysis and Modeling. Interface Design Concepts-Applying Interface Design Steps-User Interface Design Patterns-Design Issues –Design Evolution.

UNIT-V

Testing Tactics: Software Testing Fundamentals- Black -Box and White-Box Testing- White Box Testing-Basis Path Testing- Control Structure Testing: Condition Testing- Data Flow Testing-Loop Testing- Black Box Testing- Quality Concepts: Quality- Quality Control –Quality Assurance –Cost Of Quality.

TEXT BOOKS

1. Roger S. Pressman. 2010. Software Engineering – A Practitioner's Approach, 7th Edition, McGraw Hill International Edition, New Delhi.
   (Page Nos .: 34-93, 208-215, 226-232, 248-250, 259-271, 287-298, 304-306, 356-381, 420-439, 462-464)

REFERENCES

1. Ian Summerville. 2005. Software Engineering 6th Edition, Pearson Education Publication, New Delhi.
2. Daniel Hoffman and Paul Strooner. 2006. Software Design Automated Testing and Maintenance, Thomson Publications, Asia.
3. Kalkar S.A. 2007. Software Engineering a Concise Study, 1st edition, Prentice Hall of India, New Delhi.
4. Richard Fairley. 1998. Software Engineering Concepts, 1st Edition, Tata McGraw Hill Publishing, New Delhi.
5. Stephen Schach. 2007. Software Engineering, 7th Edition, Tata McGraw Hill, New Delhi.

WEB SITES

1. http://en.wikipedia.org/wiki/Software_engineering
2. http://www.onesmartclick.com/engineering/software-engineering.html
3. http://www.cc.gatech.edu/classes/AY2000/cs3802_fall/

ESE MARKS ALLOCATION

1.	**Section A** 20 x 1 = 20	20
2.	**Section B** 5 x 8 = 40 Either 'A' or 'B' choice	40
	Total	60

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

Coimbatore – 641 021.

LECTURE PLAN
DEPARTMENT OF COMPUTER SCIENCE

STAFF NAME: N. MANONMANI

SUBJECT NAME: SOFTWARE ENGINEERING SUB.CODE: 15CSU601

SEMESTER: VI CLASS: III B.Sc (CS)

S.No.	Lecture Duration	Topics to be Covered	Support Materials/Page Nos
		UNIT-I	
1.	1	Introduction to Software Engineering	T1: 12, W1, W2
2.	1	The Evolving Role of Software	T1: 3-4
3.	1	Software	T1: 4-10
4.	1	Software Myths	T1: 21-23
5.	1	A Generic View of process	T1: 31-33
6.	1	A Generic View of process: Defining a Framework Activity, Identifying a Task Set, Process Patterns	T1: 33-36
7.	1	Software Engineering, A Layered Technology	T1: 12-13
8.	1	Process Models: Prescriptive Models	T1: 38-39
9.	1	Waterfall Model	T1: 39-40
10.	1	Incremental process Models.	T1: 41-42
11.	1	Evolutionary Process Models: Prototyping	T1: 42-44
12.	1	The Spiral Model	T1: 45-47
13.	1	Specialized process Models	T1: 50-52
14.	1	Recapitulation and Discussion of important questions	

	Total No. of Hours Planned for Unit-I = 14		
		UNIT-II	
1.	1	Building the Analysis Model: Requirements Analysis:	T1: 149-150
2.	1	Requirements Analysis : Overall Objectives and Philosophy, Analysis Rules of Thumb	T1: 150-151
3.	1	Requirements Analysis: Domain Analysis, Requirements Modeling Approaches	T1: 151-153
4.	1	Analysis Modeling Approaches	T1: 153-154, W2
5.	1	Data Modeling Concepts: Data Objects	T1: 164
6.	1	Data Modeling Concepts: Data attributes	T1: 164-165
7.	1	Relationships Cardinality and Modality	T1: 165-166
8.	1	Flow Oriented Modeling: Creating Data Flow Model	T1: 187-190
9.	1	Creating a Control Flow Model	T1: 191
10.	1	The Control Specification	T1: 191-192
11.	1	The Process Specification	T1: 192-194
12.	1	Creating a Behavioral Model: Identifying Events with the Use Case	T1: 195-196
13.	1	Creating a Behavioral Model: State Representations	T1: 196-199
14.	1	Recapitulation and Discussion of important questions	
	Total No. of Hours Planned for Unit-II = 14		
		UNIT-III	
1.	1	Design Engineering: Design with the Context of Software Engineering	T1: 215-218
2.	1	Design Process and Design Quality	T1: 219-221
3.	1	Design Concepts: Abstraction, Architecture, Patterns,	T1: 222-224

4.	1	Design Concepts: Separation of Concerns, Modularity, Information Hiding	T1: 225-226
5.	1	Design Concepts: Functional Independence, Refinement, Aspects, Refactoring, Object-Oriented Design Concepts, Design Classes	T1: 227-230
6.	1	Creating An Architectural Design: Software Architecture	T1: 243-244
7.	1	Software Architecture: Architectural Descriptions, Architectural Decisions	T1: 245-246
8.	1	Data Design	T1: 234
9.	1	Architectural Design: Representing the System in Context, Defining Archetypes	T1: 255-257
10.	1	Architectural Design: Refining the Architecture into Components, Describing Instantiations of the System	T1: 258-260
11.	1	Assessing Alternative Architectural Designs	T1: 261-264
12.	1	Mapping Data Flow into Software Architecture: Transform Mapping	T1: 265-268
13.	1	Mapping Data Flow into Software Architecture: Refining the Architectural Design	T1: 269-272
14.	1	Recapitulation and Discussion of important questions	
	Total No. of Hours Planned for Unit-III = 14		
		UNIT-IV	
1.	1	Performing User Interface Design: The Golden Rules: Place the User in Control	T1: 312-313
2.	1	Reduce the User's Memory Load	T1: 314-315
3.	1	Make the Interface Consistent	T1: 316-317
4.	1	User Interface Analysis and Design: Interface Analysis and Design Models	T1: 317-318

5.	1	The Process	T1: 319
6.	1	Interface Analysis: User Analysis	T1: 320-321
7.	1	Task analysis and Modeling: Use case, Task elaboration, Object elaboration	T1: 322-324
8.	1	Task analysis and Modeling: Workflow analysis, Hierarchical representation	T1: 325-327
9.	1	Interface Design Concepts: Applying Interface Design Steps	T1: 328-329
10.	1	User Interface Design Patterns	T1: 330
11.	1	Design Issues: Response time, Help facilities	T1: 331-332
12.	1	Design Issues: Error Handling, Menu and command labeling, Application accessibility, Internalization	T1: 333-334
13.	1	Design Evaluation	T1: 342-343
14.	1	Recapitulation and Discussion of important questions	
	Total No. of Hours Planned for Unit-IV=14		
		UNIT-V	
1.	1	Testing Tactics: Software Testing Fundamentals	T1: 482-483
2.	1	Black Box and White Box Testing	T1: 485, 495
3.	1	White Box Testing	T1: 485
4.	1	Basis Path Testing: Flow Graph Notation, Independent Program Paths	T1: 485-488
5.	1	Basis Path Testing: Deriving Test Cases, Graph Matrices	T1: 489-491
6.	1	Control Structure Testing	T1: 492
7.	1	Data Flow Testing	T1: 493
8.	1	Loop Testing	T1: 493
9.	1	Black Box Testing: Graph-Based Testing Methods, Equivalence Partitioning	T1: 495-497

10.	1	Black Box Testing: Boundary Value Analysis, Orthogonal Array Testing	T1: 498-501
11.	1	Quality Concepts	T1: 398-399
12.	1	Quality	T1: 400-405
13.	1	Quality Control	T1: 412
14.	1	Quality Assurance	T1: 413
15.	1	Cost Of Quality	T1: 407-408
16.	1	Recapitulation and Discussion of important questions	
17.	1	Recapitulation and Discussion of previous semester question papers	
18.	1	Recapitulation and Discussion of previous semester question papers	
19.	1	Recapitulation and Discussion of previous semester question papers	
	Total No. of Hours Planned for Unit-V = 19		
Total Planned Hours	75		

TEXT BOOKS

1. Roger S. Pressman. 2010. Software Engineering – A Practitioner's Approach, 7th Edition, McGraw Hill International Edition, New Delhi.
(Page Nos .: 34-93, 208-215, 226-232, 248-250, 259-271, 287-298, 304-306, 356-381, 420-439, 462-464)

REFERENCES

1. Ian Summerville. 2005. Software Engineering 6th Edition, Pearson Education Publication, New Delhi.Daniel Hoffman and Paul Strooner. 2006. Software Design Automated Testing and Maintenance, Thomson Publications, Asia.
2. Kalkar S.A. 2007. Software Engineering a Concise Study, 1st edition, Prentice Hall of India, New Delhi.
3. Richard Fairley. 1998. Software Engineering Concepts, 1st Edition, Tata McGraw Hill Publishing, New Delhi.
4. Stephen Schach. 2007. Software Engineering, 7th Edition, Tata McGraw Hill, New Delhi.

WEB SITES

W1: http://en.wikipedia.org/wiki/Software_engineering

W2: http://www.onesmartclick.com/engineering/software-engineering.html

W3: http://www.cc.gatech.edu/classes/AY2000/cs3802_fall/

UNIT-I

SYLLABUS

Introduction to Software Engineering: The Evolving Role of Software-Software-Software Myths- A Generic View of process: Software Engineering –A Layered Technology- Process Models: Prescriptive Models- Waterfall Model- Incremental process Models. Evolutionary Process Models: Prototyping, The Spiral Model. Specialized process Models

INTRODUCTION TO SOFTWARE ENGINEERING

Computer software is the product that software professionals build and then support over the long term. It encompasses programs that execute within a computer of any size and architecture, content that is presented as the computer programs execute, and descriptive information in both hard copy and virtual forms that encompass virtually any electronic media.

Software engineering encompasses a process, a collection of methods (practice) and an array of tools that allow professionals to build high quality computer software. Software engineering is important because it enables us to build complex systems in a timely manner and with high quality.

Software Engineering

Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

Software project management has wider scope than software engineering process as it involves communication, pre and post delivery support etc

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures.

The outcome of software engineering is an efficient and reliable software product.

Definitions

IEEE defines software engineering as:

- The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.

The study of approaches as in the above statement, Fritz Bauer, a German computer scientist, defines software engineering as:

- Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and work efficiently on real machines.

Software engineering is about teams. The problems to solve are so complex or large, that a single developer cannot solve them anymore. Software engineering is also about communication. Teams do not consist only of developers, but also of testers, architects, system engineers, customer, project managers, etc.

Software projects can be so large that needs careful planning. Implementation is no longer just writing code, but it is also following guidelines, writing documentation and also writing unit tests. But unit tests alone are not enough.

The different pieces have to fit together. Problematic areas have to be spotted using metrics. They tell us if our code follows certain standards. Once coding is finished, that does not mean that the project is finished: for large projects maintaining software can keep many people busy for a long time.

Since there are so many factors influencing the success or failure of a project, there is a need to learn a little about project management and its pitfalls, but especially what makes projects successful. And last but not least, a good software engineer, like any engineer, needs tools, and to know about them is important.

Developers Work in Teams

In beginning coding was done by individuals. The problems solved earlier were small enough so one person could master them. In the real world this is different:- the problem sizes and time constraints are such that only teams can solve those problems.

For teams to work effectively they need a language to communicate (UML). Also teams do not consist only of developers, but also of testers, architects, system engineers and most

importantly the customer. There is a need to learn about what makes good teams, how to communicate with the customer, and how to document not only the source code, but everything related to the software project.

New Language

Programming languages, such as Java or C++, was used earlier and turn ideas into code. But these ideas are independent of the language. Unified Modeling Language (UML) is a way to describe code independently of language, and more importantly, it helps to think in one higher level of abstraction. UML can be an invaluable communication and documentation tool.

Pattern gives one higher level of abstraction. Again this increases our vocabulary to communicate more effectively with our peers. Also, it is a fantastic way to learn from our seniors. This is essential for designing large software systems.

Measurement

Also just being able to write software, doesn't mean that the software is any good. Discovering what makes good software, and how to measure software quality is necessary. Analysis of existing source code through static analysis and measuring metrics is needed.

It is needed to ensure that the code meets certain quality standards. Testing is also important in this context, it guarantees high quality products.

New Tools

Apart from an IDE, a compiler and a debugger, there are many more tools at the disposal of a software engineer. There are tools that allow us to work in teams, to document our software, to assist and monitor the whole development effort. There are tools for software architects, tools for testing and profiling, automation and re-engineering.

EVOLVING ROLE OF SOFTWARE

The industry originated with the entrepreneurial computer software and services companies of the 1950s and 1960s, grew dramatically through the 1970s and 1980s to become a market force rivaling that of the computer hardware companies, and by the 1990s had become the supplier of technical know-how that transformed the way people worked, played and communicated every day of their lives. The following are the different eras' of software engineering:

The Pioneering Era (1955-1965)

The most important development was that new computers were coming out almost every year or two, rendering existing ones obsolete. Software people had to rewrite all their programs to run on these new machines.

Jobs were run by signing up for machine time or by operational staff by putting punched cards for input into the machine's card reader and waiting for results to come back on the printer.

The field was so new that the idea of management by schedule was non-existent. Making predictions of a project's completion date was almost impossible.

Computer hardware was application-specific. Scientific and business tasks needed different machines.

Hardware vendors gave away systems software for free as hardware could not be sold without software. A few companies sold the service of building custom software but no software companies were selling packaged software.

The Stabilizing Era (1965-1980)

The whole job-queue system had been institutionalized and so programmers no longer ran their jobs except for peculiar applications like on-board computers. To handle the jobs, an enormous bureaucracy had grown up around the central computer center.

The major problem as a result of this bureaucracy was turnaround time, the time between job submission and completion. At worst it was measured in days.

Then came IBM 360. It signaled the beginning of the stabilizing era. This was the largest software project to date. The 360 also combined scientific and business applications onto one machine.

The job control language (JCL) raised a whole new class of problems. The programmer had to write the program in a whole new language to tell the computer and OS what to do. JCL was the least popular feature of the 360.

"Structured Programming" burst on the scene in the middle of this era. PL/I, introduced by IBM to merge all programming languages into one, failed. Most customized applications continued to be done in-house.

The Micro Era (1980-Present)

The price of computing has dropped dramatically making ubiquitous computing possible. Now every programmer can have a computer on his desk. The old JCL has been replaced by the user friendly GUI.

The software part of the hardware architecture that the programmer must know about, such as the instruction set, has not changed much since the advent of the IBM mainframe and the first Intel chip.

The most-used programming languages today are between 15 and 40 years old. The Fourth Generation Languages never achieved the dream of "programming without programmers" and the idea is pretty much limited to report generation from databases. There is an increasing clamor though for more and better software research.

Computer software continues to be the single most important technology on the world stage. And it’s also a prime example of the law of unintended consequences.

Evolution of software on Different Industries:

- Fifty years ago no one could have predicted that software would become an indispensable technology for business, science, and engineering; that software would enable the creation of new technologies (e.g., genetic engineering and nanotechnology), the extension of existing technologies (e.g., telecommunications), and the radical change in older technologies (e.g., the printing industry).
- Software would be the driving force behind the personal computer revolution; that shrink-wrapped software products would be purchased by consumers in neighborhood malls; that software would slowly evolve from a product to a service as "on-demand" software companies deliver just-in-time functionality via a Web browser;
- A software company would become larger and more influential than almost all industrial-era companies; that a vast software-driven network called the Internet would evolve and change everything from library research to consumer shopping to political discourse to the dating habits of young (and not so young) adults.
- As software's importance has grown, the software community has continually attempted to develop technologies that will make it easier, faster, and less expensive to build and maintain high-quality computer programs. Some of these technologies are targeted at a specific application domain (e.g., website design and implementation); others focus on a technology domain (e.g., object-oriented systems or aspect oriented programming); and still others are broad-based (e.g., operating systems such as Linux).
- However, we have yet to develop a software technology that does it all, and the likelihood of one arising in the future is small. And yet, people bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right.

SOFTWARE

Software is: (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

SOFTWARE CHARACTERISTICS

Software is developed or engineered; it is not manufactured in the classical sense.

Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software.

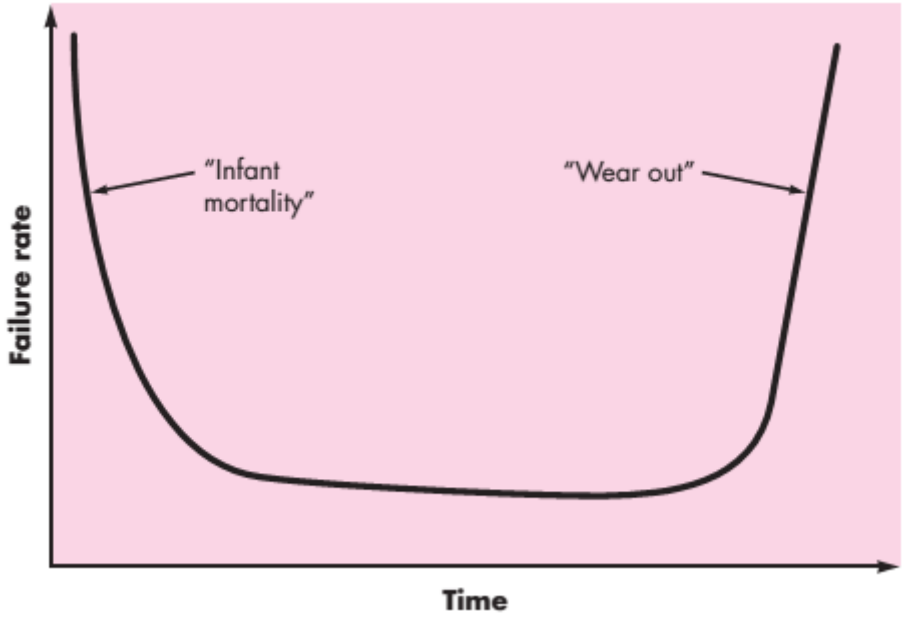


Fig 1.1. Failure curve for hardware

Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different. Both activities require the construction of a "product," but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

Software doesn't "wear out." Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to *wear out.*

Software is not susceptible to the environmental problems that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure 1.2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does *deteriorate.*

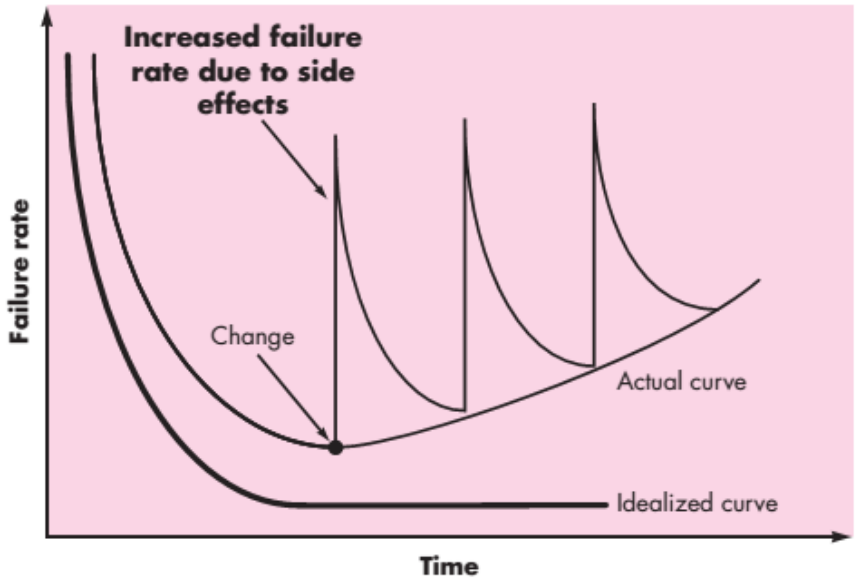


Fig 1.2. Failure curves for software

This seeming contradiction can best be explained by considering the actual curve in Figure 1.2. During its life, software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve" (Figure 1.2). Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts.

- Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.

Although the industry is moving toward component-based construction, most software continues to be custom built. As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new. In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

- **A software component** should be designed and implemented so that it can be reused in many different programs.
- Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts.
- For example, today's interactive user interfaces are built with reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

Software Application Domains

Today, seven broad categories of computer software present continuing challenges for software engineers:

System software—a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate,4 information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data. In either case, the systems software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

Application software—stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making. In addition to conventional data processing applications, application software is used to control business functions in real time (e.g., point-of-sale transaction processing, real-time manufacturing process control).

Engineering/scientific software—has been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

Embedded software—resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. Embedded software can perform limited and esoteric functions (e.g., key pad control for a microwave oven) or provide significant

function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

Product-line software—designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications).

Web applications—called "WebApps," this network-centric software category spans a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics. However, as Web 2.0 emerges, WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

Artificial intelligence software—makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

Millions of software engineers worldwide are hard at work on software projects in one or more of these categories. In some cases, new systems are being built, but in many others, existing applications are being corrected, adapted, and enhanced. It is not uncommon for a young software engineer to work a program that is older than she is! Past generations of software people have left a legacy in each of the categories I have discussed. Hopefully, the legacy to be left behind by this generation will ease the burden of future software engineers. And yet, new challenges (Chapter 31) have appeared on the horizon:

Open-world computing—the rapid growth of wireless networking may soon lead to true pervasive, distributed computing. The challenge for software engineers will be to develop systems and application software that will allow mobile devices, personal computers, and enterprise systems to communicate across vast networks

Netsourcing—the World Wide Web is rapidly becoming a computing engine as well as a content provider. The challenge for software engineers is to architect simple (e.g., personal financial planning) and sophisticated applications that provide a benefit to targeted end-user markets worldwide.

Open source—a growing trend that results in distribution of source code for systems applications (e.g., operating systems, database, and development environments) so that many

people can contribute to its development. The challenge for software engineers is to build source code that is self-descriptive, but more importantly, to develop techniques that will enable both customers and developers to know what changes have been made and how those changes manifest themselves within the software.

Each of these new challenges will undoubtedly obey the law of unintended consequences and have effects (for businesspeople, software engineers, and end users) that cannot be predicted today. However, software engineers can prepare by instantiating a process that is agile and adaptable enough to accommodate dramatic changes in technology and to business rules that are sure to come over the next decade.

Legacy Software

Hundreds of thousands of computer programs fall into one of the seven broad application domains discussed in the preceding subsection. Some of these are state of-the-art software—just released to individuals, industry, and government. But other programs are older, in some cases *much* older.

These older programs—often referred to as *legacy software*—have been the focus of continuous attention and concern since the 1960s.

Unfortunately, there is sometimes one additional characteristic that is present in legacy software—*poor quality*.

However, as time passes, legacy systems often evolve for one or more of the following reasons:

The software must be adapted to meet the needs of new computing environments or technology.

The software must be enhanced to implement new business requirements.

The software must be extended to make it interoperable with other more modern systems or databases.

The software must be re-architected to make it viable within a network environment.

When these modes of evolution occur, a legacy system must be reengineered so that it remains viable into the future. The goal of modern software engineering is to "devise methodologies that are founded on the notion of evolution"; that is, the notion that software systems continually change, new software systems are built from the old ones, and . . . all must interoperate and cooperate with each other.

SOFTWARE MYTHS

Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing.

Myths have a number of attributes that make them insidious. For instance, they appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score."

Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike. However, old attitudes and habits are difficult to modify, and remnants of software myths remain.

Management myths.

Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

Myth: *We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?*

Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

Myth: *If we get behind schedule, we can add more programmers and catch up (sometimes called the "Mongolian horde" concept).*

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks [Bro95]: "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well coordinated manner.

Myth: *If I decide to outsource the software project to a third party, I can just relax and let that firm build it.*

Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

Customer myths.

A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer.

Myth: *A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.*

Reality: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous

requirements (usually derive iteratively) are developed only through effective and continuous communication between customer and developer.

Myth: *Software requirements continually change, but change can be easily accommodated because software is flexible.*

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small.16 However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

Practitioner's myths.

Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

Myth: *Once we write the program and get it to work, our job is done.*

Reality: Someone once said that "the sooner you begin 'writing code,' the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: *Until I get the program "running" I have no way of assessing its quality.*

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—*the technical review.* Software reviews (described in Chapter 15) are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

Myth: *The only deliverable work product for a successful project is the working program.*

Reality: A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.

Myth: *Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.*

Reality: Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times

Many software professionals recognize the fallacy of the myths just described. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

A GENERIC VIEW OF PROCESS

A process is defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.

The software process is represented schematically in Figure 1.3. Referring to the figure, each framework activity is populated by a set of software engineering actions.

Each software engineering action is defined by a *task set* that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

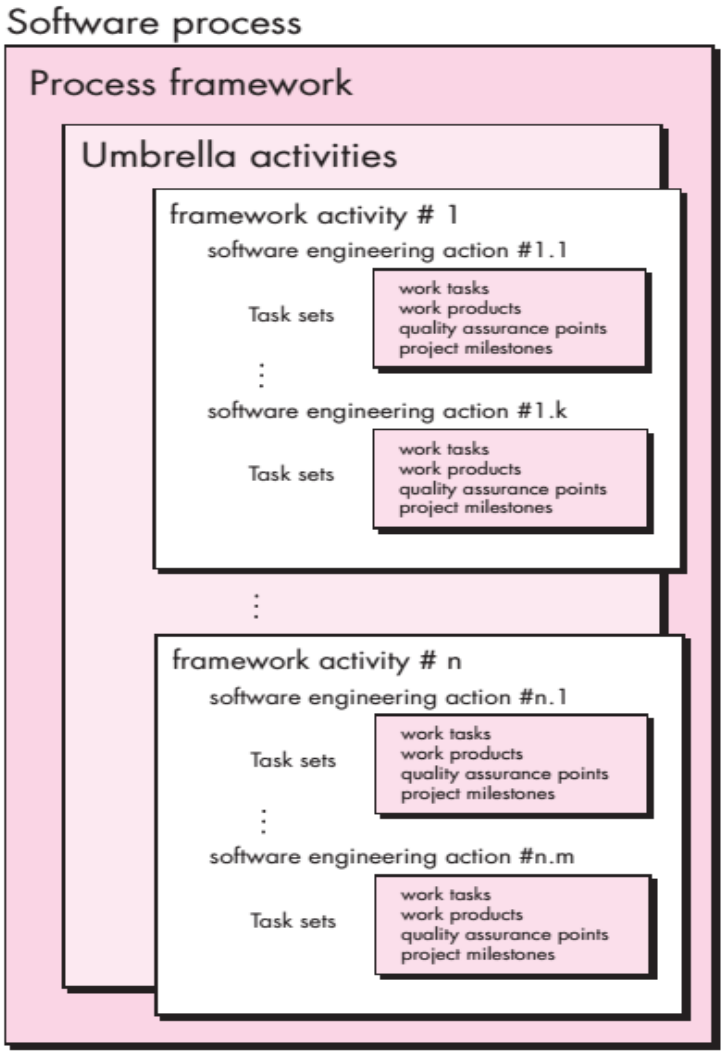


Fig 1.3. A software process framework

A generic process framework for software engineering defines five framework activities—**communication, planning, modeling, construction,** and **deployment.** In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.

One important aspect of the software process is called *process flow*—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in Figure 1.4.

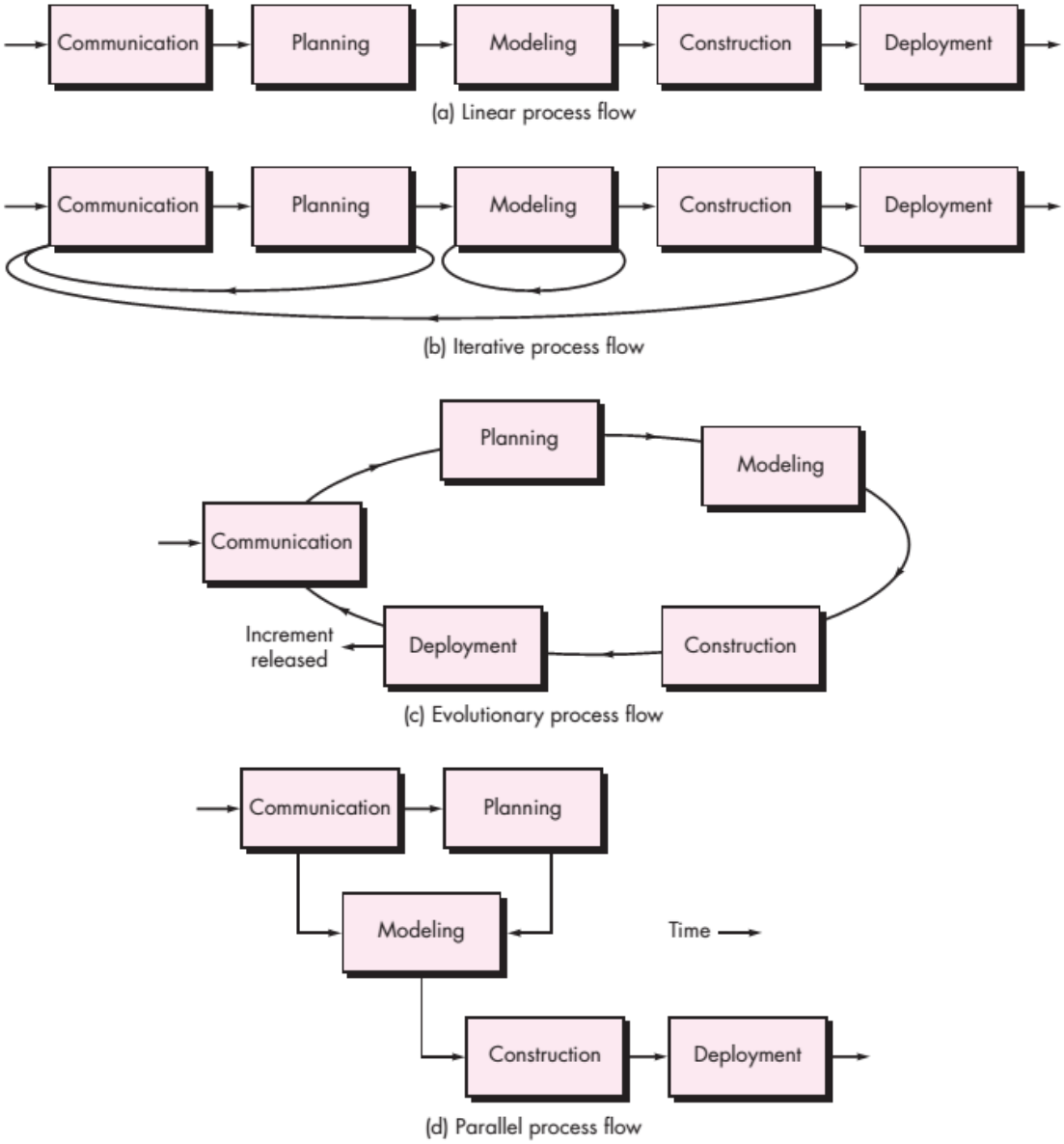


Fig 1.4. Process flow

- A *linear process flow* executes each of the five framework activities in sequence, beginning with communication and culminating with deployment (Figure 1.4a).

- An *iterative process flow* repeats one or more of the activities before proceeding to the next (Figure 1.4.b).
- An *evolutionary process flow* executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software (Figure 1.4c).
- A *parallel process flow* (Figure 1.4d) executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software)

Defining a Framework Activity

A software team would need significantly more information before it could properly execute any one of these activities as part of the software process.

A key question is: *What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project*

For a small software project requested by one person (at a remote location) with simple, straightforward requirements, the communication activity might encompass little more than a phone call with the appropriate stakeholder. Therefore, the only necessary action is *phone conversation,* and the work tasks (the *task set*) that this action encompasses are:

1. Make contact with stakeholder via telephone.

2. Discuss requirements and take notes.

3. Organize notes into a brief written statement of requirements.

4. E-mail to stakeholder for review and approval.

If the project was considerably more complex with many stakeholders, each with a different set of (sometime conflicting) requirements, the communication activity might have six distinct actions *inception, elicitation, elaboration, negotiation, specification,* and *validation.* Each of these software engineering actions would have many work tasks and a number of distinct work products.

Identifying a Task Set

Referring again to Figure 1.3 each software engineering action (e.g., *elicitation,* an action associated with the communication activity) can be represented by a

- Number of different *task sets*—each a collection of software engineering work tasks,
- Related work products,
- Quality assurance points,
- Project milestones.

Choose a task set that best accommodates the needs of the project and the characteristics of your team. This implies that a software engineering action can be adapted to the specific needs of the software project and the characteristics of the project team.

Task Set

A task set defines the actual work to be done to accomplish the objectives of a software engineering action. For example, *elicitation* (more commonly called "requirements gathering") is an important software engineering action that occurs during the communication activity. The goal of requirements gathering is to understand what various stakeholders want from the software that is to be built. For a small, relatively simple project, the task set for requirements gathering might look like this:

1. Make a list of stakeholders for the project.
2. Invite all stakeholders to an informal meeting.
3. Ask each stakeholder to make a list of features and functions required.
4. Discuss requirements and build a final list.
5. Prioritize requirements.
6. Note areas of uncertainty.

Process Patterns

Every software team encounters problems as it moves through the software process. It would be useful if proven solutions to these problems were readily available to the team so that the problems could be addressed and resolved quickly.

A *process pattern* describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem.

Stated in more general terms, a process pattern provides you with a template—a consistent method for describing problem solutions within the context of the software process. By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.

Patterns can be defined at any level of abstraction. In some cases, a pattern might be used to describe a problem (and solution) associated with a complete process model (e.g., prototyping). In other situations, patterns can be used to describe a problem (and solution) associated with a framework activity (e.g., **planning**) or an action within a framework activity (e.g., project estimating). Ambler has proposed a template for describing a process pattern:

Pattern Name. The pattern is given a meaningful name describing it within the context of the software process (e.g., **TechnicalReviews**).

Forces. The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

Type. The pattern type is specified. Ambler [Amb98] suggests three types:

1. *Stage pattern*—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity). An example of a stage pattern might be **EstablishingCommunication.** This pattern would incorporate the task pattern **RequirementsGathering** and others.

2. *Task pattern*—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., **RequirementsGathering** is a task pattern).

3. *Phase pattern*—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature. An example of a phase pattern might be **SpiralModel** or **Prototyping.**

Initial context. Describes the conditions under which the pattern applies. Prior to the initiation of the pattern: (1) What organizational or team-related activities have already occurred? (2) What is the entry state for the process? (3) What software engineering information or project information already exists? For example, the **Planning** pattern (a stage pattern) requires that (1) customers and software engineers have established a collaborative communication; (2) successful completion of a number of task patterns [specified] for the

Communication pattern has occurred; and (3) the project scope, basic business requirements, and project constraints are known.

Problem. The specific problem to be solved by the pattern.

Solution. Describes how to implement the pattern successfully. This section describes how the initial state of the process (that exists before the pattern is implemented) is modified as a consequence of the initiation of the pattern. It also describes how software engineering information or project information that is available before the initiation of the pattern is transformed as a consequence of the successful execution of the pattern.

Resulting Context. Describes the conditions that will result once the pattern has been successfully implemented. Upon completion of the pattern: (1) What organizational or team-related activities must have occurred? (2) What is the exit state for the process? (3) What software engineering information or project information has been developed?

Related Patterns. Provide a list of all process patterns that are directly related to this one. This may be represented as a hierarchy or in some other diagrammatic form. For example, the stage pattern **Communication** encompasses the task patterns: ProjectTeam, CollaborativeGuidelines, ScopeIsolation, RequirementsGathering, ConstraintDescription, and ScenarioCreation**.**

Known Uses and Examples. Indicate the specific instances in which the pattern is applicable. For example, **Communication** is mandatory at the beginning of every software project, is recommended throughout the software project, and is mandatory once the deployment activity is under way.

Process patterns provide an effective mechanism for addressing problems associated with any software process. The patterns enable you to develop a hierarchical process description that begins at a high level of abstraction (a phase pattern).

The description is then refined into a set of stage patterns that describe framework activities and are further refined in a hierarchical fashion into more detailed task patterns for each stage pattern. Once process patterns have been developed, they can be reused for the definition of process variants—that is, a customized process model can be defined by a software team using the patterns as building blocks for the process model.

SOFTWARE ENGINEERING

In order to build software that is ready to meet the challenges of the twenty-first century, you must recognize a few simple realities:

- ***It follows that a concerted effort should be made to understand the problem before a software solution is developed.***

• Software has become deeply embedded in virtually every aspect of our lives, and as a consequence, the number of people who have an interest in the features and functions provided by a specific application has grown dramatically. When a new application or embedded system is to be built, many voices must be heard. And it sometimes seems that each of them has a slightly different idea of what software features and functions should be delivered.

- ***It follows that design becomes a pivotal activity.***

• The information technology requirements demanded by individuals, businesses, and governments grow increasing complex with each passing year. Large teams of people now create computer programs that were once built by a single individual. Sophisticated software that was once implemented in a predictable, self-contained, computing environment is now embedded inside everything from consumer electronics to medical devices to weapons systems. The complexity of these new computer-based systems and products demands careful attention to the interactions of all system elements.

- ***It follows that software should exhibit high quality.***

• Individuals, businesses, and governments increasingly rely on software for strategic and tactical decision making as well as day-to-day operations and control. If the software fails, people and major enterprises can experience anything from minor inconvenience to catastrophic failures.

- ***It follows that software should be maintainable.***

• As the perceived value of a specific application grows, the likelihood is that its user base and longevity will also grow. As its user base and time-in-use increase, demands for adaptation and enhancement will also grow.

These simple realities lead to one conclusion: *software in all of its forms and across all of its application domains should be engineered.* And that leads us to the topic of this book—*software engineering.*

Although hundreds of authors have developed personal definitions of software engineering, a definition proposed by Fritz Bauer [Nau69] at the seminal conference on the subject still serves as a basis for discussion:

- **[Software engineering is]** the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.
- **Software Engineering:** (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1). And yet, a "systematic, disciplined, and quantifiable" approach applied by one software team may be burdensome to another. We need discipline, but we also need adaptability and agility.

A LAYERED TECHNOLOGY

Software engineering is a layered technology. Referring to Figure 1.5, any engineering approach (including software engineering) must rest on an organizational commitment to quality.

Total quality management, Six Sigma, and similar philosophies10 foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering.

The bedrock that supports software engineering is a quality focus. The foundation for software engineering is the *process* layer.

The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software.

Fig 1.5 Software engineering layers

Process defines a framework that must be established for effective delivery of software engineering technology.

The software process forms the basis for

- Management control of software projects and establishes the context in which technical methods are applied
- Work products (models, documents, data, reports, forms, etc.) are produced

- Milestones are established
- Quality is ensured
- Change is properly managed.

Software engineering *methods* provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Software engineering *tools* provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called *computer-aided software engineering*, is established.

PROCESS MODELS:

PRESCRIPTIVE MODELS

Prescriptive process models were originally proposed to bring order to the chaos of software development. History has indicated that these traditional models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams. However, software engineering work and the product that it produces remain on "the edge of chaos."

Change occurs when there is some structure so that the change can be organized, but not so rigid that it cannot occur. Too much chaos, on the other hand, can make coordination and coherence impossible. Lack of structure does not always mean disorder. The philosophical implications of this argument are significant for software engineering.

If prescriptive process models strive for structure and order, are they inappropriate for a software world that thrives on change? Yet, if we reject traditional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

There are no easy answers to these questions, but there are alternatives available to software engineers. In the sections that follow, I examine the prescriptive process approach in which order and project consistency are dominant issues. I call them "prescriptive" because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a *work flow*)—that is, the manner in which the process elements are interrelated to one another.

WATERFALL MODEL

There are times when the requirements for a problem are well understood—when work flows from **communication** through **deployment** in a reasonably linear fashion. This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made (e.g., an adaptation to accounting software that has been mandated because of

changes to government regulations). It may also occur in a limited number of new development efforts, but only when requirements are well defined and reasonably stable.

The *waterfall model*, sometimes called the *classic life cycle*, suggests a systematic, sequential approach6 to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software (Figure 1.6).

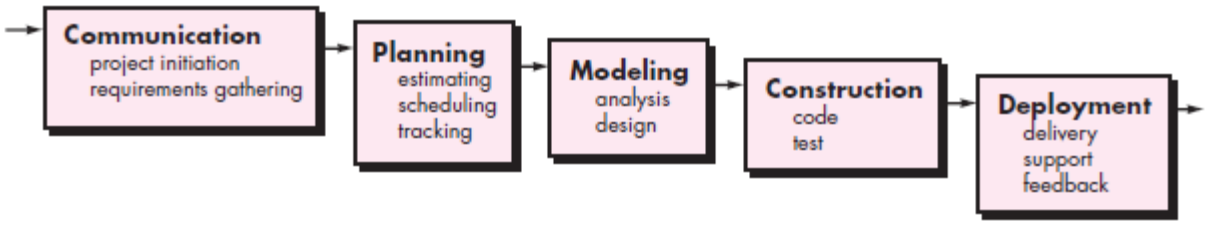


Fig 1.6.The waterfall model

The waterfall model is the oldest paradigm for software engineering. However, over the past three decades, criticism of this process model has caused even passionate supporters to question its efficacy. Among the problems that are sometimes encountered when the waterfall model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

In an interesting analysis of actual projects, Bradac found that the linear nature of the classic life cycle leads to "blocking states" in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking states tend to be more prevalent at the beginning and end of a linear sequential process.

Today, software work is fast-paced and subject to a never-ending stream of changes (to features, functions, and information content). The waterfall model is often inappropriate for such work. However, it can serve as a useful process model in situations where requirements are fixed and work is to proceed to completion in a linear manner.

INCREMENTAL PROCESS MODELS

There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process. In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases. In such cases, you can choose a process model that is designed to produce the software in increments.

The *incremental* model combines elements of linear and parallel process. Referring to Figure 1.7, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software in a manner that is similar to the increments produced by an evolutionary process flow.

For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm.

When an incremental model is used, the first increment is often a *core product.* That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation).

As a result of use and/or evaluation, a plan is developed for the next increment.

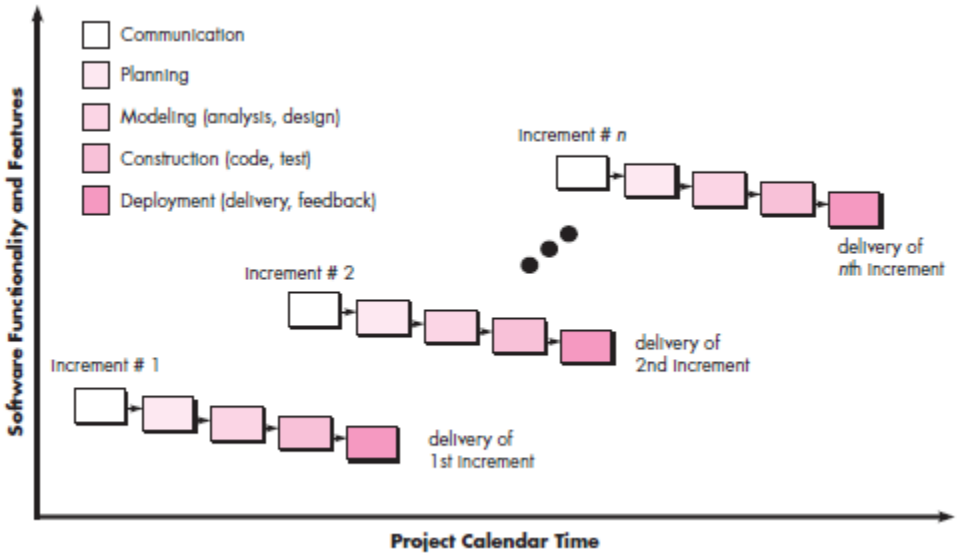


Fig 1.7. The incremental model

The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment.

In addition, increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end users without inordinate delay.

EVOLUTIONARY PROCESS MODELS

Software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight line path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined.

In these and similar situations, a process model that has been explicitly designed to accommodate a product that evolves over time is needed. Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software.

PROTOTYPING

Prototyping. Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a *prototyping paradigm* may offer the best approach.

Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models. Regardless of the manner in which it is applied, the prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy.

The prototyping paradigm (Figure 2.6) begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A prototyping iteration is planned quickly, and modeling (in the form of a "quick design") occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats).

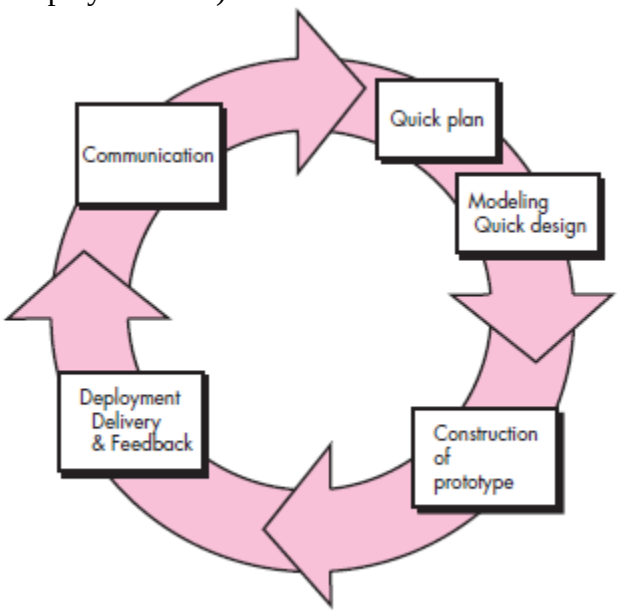


Fig 1.8. The prototyping paradigm

The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built, you can make use of existing program fragments or apply tools (e.g., report generators and window managers) that enable working programs to be generated quickly.

In most projects, the first system built is barely usable. It may be too slow, too big, awkward in use or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved.

The prototype can serve as "the first system." Although some prototypes are built as "throwaways," others are evolutionary in the sense that the prototype slowly evolves into the actual system.

Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately. Yet, prototyping can be problematic for the following reasons:

1. Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, stakeholders cry foul and demand that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.

2. As a software engineer, you often make implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

Although problems can occur, prototyping can be an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, all stakeholders should agree that the prototype is built to serve as a mechanism for defining requirements. It is then discarded (at least in part), and the actual software is engineered with an eye toward quality.

THE SPIRAL MODEL

The Spiral Model. Originally proposed by Barry Boehm, the *spiral model* is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software. Boehm describes the model in the following manner:

The spiral development model is a *risk*-driven *process model* generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a *cyclic* approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of *anchor point milestones* for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

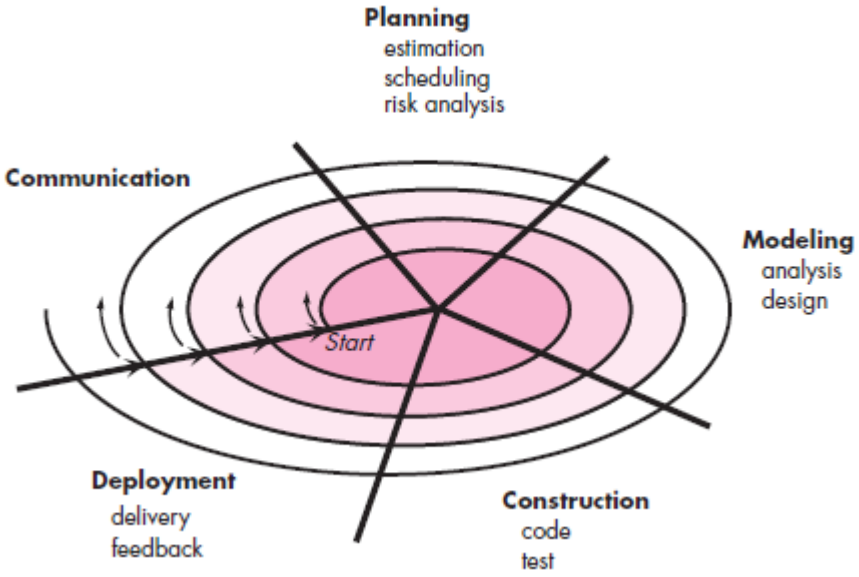


Fig 1.9. A typical spiral model

A spiral model is divided into a set of framework activities defined by the software engineering team. For illustrative purposes, I use the generic framework activities discussed earlier. Each of the framework activities represent one segment of the spiral path illustrated in Figure 1.9. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk is considered as each revolution is made. *Anchor point milestones*—a combination of work products and conditions that are attained along the path of the spiral—are noted for each evolutionary pass.

The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.

Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. Therefore, the first circuit around the spiral might represent a "concept development project" that starts at the core of the spiral and continues for multiple iterations until concept development is complete.

If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "new product development project" commences. The new product will evolve through a number of iterations around the spiral. Later, a circuit around the spiral might be used to represent a "product enhancement project." In essence, the spiral, when characterized in this way, remains operative until the software is retired. There are times when the process is dormant,

but whenever a change is initiated, the process starts at the appropriate entry point (e.g., product enhancement).

The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level.

The spiral model uses prototyping as a risk reduction mechanism but, more important, enables you to apply the prototyping approach at any stage in the evolution of the product. It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world. The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.

But like other paradigms, the spiral model is not a panacea. It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur.

A Final Word on Evolutionary Processes

Modern computer software is characterized by continual change, by very tight time lines, and by an emphatic need for customer–user satisfaction. In many cases, time-to-market is the most important management requirement. If a market window is missed, the software project itself may be meaningless.

Evolutionary process models were conceived to address these issues, and yet, as a general class of process models, they too have weaknesses. Despite the unquestionable benefits of evolutionary software processes, we have some concerns. The first concern is that prototyping [and other more sophisticated evolutionary processes] poses a problem to project planning because of the uncertain number of cycles required to construct the product. Most project management and estimation techniques are based on linear layouts of activities, so they do not fit completely.

Second, evolutionary software processes do not establish the maximum speed of the evolution. If the evolutions occur too fast, without a period of relaxation, it is certain that the process will fall into chaos. On the other hand if the speed is too slow then productivity could be affected.

Third, software processes should be focused on flexibility and extensibility rather than on high quality. This assertion sounds scary. However, we should prioritize the speed of the development over zero defects. Extending the development in order to reach high quality could result in a late delivery of the product, when the opportunity niche has disappeared. This paradigm shift is imposed by the competition on the edge of chaos.

Indeed, a software process that focuses on flexibility, extensibility, and speed of development over high quality does sound scary. And yet, this idea has been proposed by a number of well-respected software engineering experts.

The intent of evolutionary models is to develop high-quality software14 in an iterative or incremental manner. However, it is possible to use an evolutionary process to emphasize

flexibility, extensibility, and speed of development. The challenge for software teams and their managers is to establish a proper balance between these critical project and product parameters and customer satisfaction (the ultimate arbiter of software quality).

SPECIALIZED PROCESS MODELS

Specialized process models take on many of the characteristics of one or more of the traditional models presented in the preceding sections. However, these models tend to be applied when a specialized or narrowly defined software engineering approach is chosen.

Component-Based Development

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built. The *component-based development model* incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from prepackaged software components.

Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

1. Available component-based products are researched and evaluated for the application domain in question.

2. Component integration issues are considered.

3. A software architecture is designed to accommodate the components.

4. Components are integrated into the architecture.

5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Your software engineering team can achieve a reduction in development cycle time as well as a reduction in project cost if component reuse becomes part of your culture

The Formal Methods Model

The *formal methods model* encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A

variation on this approach, called *cleanroom software engineering*, is currently applied by some software development organizations.

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily—not through ad hoc review, but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable you to discover and correct errors that might otherwise go undetected.

Although not a mainstream approach, the formal methods model offers the promise of defect-free software. Yet, concern about its applicability in a business environment has been voiced:

• The development of formal models is currently quite time consuming and expensive.

• Because few software developers have the necessary background to apply formal methods, extensive training is required.

• It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

These concerns notwithstanding, the formal methods approach has gained adherents among software developers who must build safety-critical software (e.g., developers of aircraft avionics and medical devices) and among developers that would suffer severe economic hardship should software errors occur.

Aspect-Oriented Software Development

Regardless of the software process that is chosen, the builders of complex software invariably implement a set of localized features, functions, and information content. These localized software characteristics are modeled as components (e.g., objectoriented classes) and then constructed within the context of a system architecture.

As modern computer-based systems become more sophisticated (and complex), certain *concerns*—customer required properties or areas of technical interest—span the entire architecture. Some concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect functions (e.g., the application of business rules), while others are systemic (e.g., task synchronization or memory management).

When concerns cut across multiple system functions, features, and information, they are often referred to as *crosscutting concerns. Aspectual requirements* define those crosscutting concerns that have an impact across the software architecture.

Aspect-oriented software development (AOSD), often referred to as *aspect-oriented programming* (AOP), is a relatively new software engineering paradigm that provides a process

and methodological approach for defining, specifying, designing, and constructing *aspects*—"mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern".

Aspect-oriented component engineering (AOCE):

AOCE uses a concept of horizontal slices through vertically-decomposed software components, called "aspects," to characterize cross-cutting functional and non-functional properties of components. Common, systemic aspects include user interfaces, collaborative work, distribution, persistency, memory management, transaction processing, security, integrity and so on.

Components may provide or require one or more "aspect details" relating to a particular aspect, such as a viewing mechanism, extensible affordance and interface kind (user interface aspects); event generation, transport and receiving (distribution aspects); data store/retrieve and indexing (persistency aspects); authentication, encoding and access rights (security aspects); transaction atomicity, concurrency control and logging strategy (transaction aspects); and so on. Each aspect detail has a number of properties, relating to functional and/or non-functional characteristics of the aspect detail.

A distinct aspect-oriented process has not yet matured. However, it is likely that such a process will adopt characteristics of both evolutionary and concurrent process models. The evolutionary model is appropriate as aspects are identified and then constructed. The parallel nature of concurrent development is essential because aspects are engineered independently of localized software components and yet, aspects have a direct impact on these components. Hence, it is essential to instantiate asynchronous communication between the software process activities applied to the engineering and construction of aspects and components.

POSSIBLE QUESTIONS
PART – B

1. Explain the different phases involved in waterfall life cycle. Give the reasons for the Failure of Water Fall Model.
2. Discuss on various types of software myths and the true aspects of the myths.
3. Explain about the Generic view of process in detail.
4. Elucidate the process model that combines the elements of waterfall and iterative fashion.
5. Explain the process model which is useful when staffing is unavailable to complete implementation.
6. Explain about the Evolutionary Process Model
7. Describe the Prescriptive process model in detail.
8. Explain with diagram the layered technology of software process along with its characteristics.
9. Explicate how the specialized models applied for software engineering approaches.

UNIT-I

SYLLABUS

Introduction to Software Engineering: The Evolving Role of Software-Software-Software Myths- A Generic View of process: Software Engineering –A Layered Technology- Process Models: Prescriptive Models- Waterfall Model- Incremental process Models. Evolutionary Process Models: Prototyping, The Spiral Model. Specialized process Models

INTRODUCTION TO SOFTWARE ENGINEERING

Computer software is the product that software professionals build and then support over the long term. It encompasses programs that execute within a computer of any size and architecture, content that is presented as the computer programs execute, and descriptive information in both hard copy and virtual forms that encompass virtually any electronic media.

Software engineering encompasses a process, a collection of methods (practice) and an array of tools that allow professionals to build high quality computer software. Software engineering is important because it enables us to build complex systems in a timely manner and with high quality.

Software Engineering

Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

Software project management has wider scope than software engineering process as it involves communication, pre and post delivery support etc

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures.

The outcome of software engineering is an efficient and reliable software product.

Definitions

IEEE defines software engineering as:

- The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.

The study of approaches as in the above statement, Fritz Bauer, a German computer scientist, defines software engineering as:

- Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and work efficiently on real machines.

Software engineering is about teams. The problems to solve are so complex or large, that a single developer cannot solve them anymore. Software engineering is also about communication. Teams do not consist only of developers, but also of testers, architects, system engineers, customer, project managers, etc.

Software projects can be so large that needs careful planning. Implementation is no longer just writing code, but it is also following guidelines, writing documentation and also writing unit tests. But unit tests alone are not enough.

The different pieces have to fit together. Problematic areas have to be spotted using metrics. They tell us if our code follows certain standards. Once coding is finished, that does not mean that the project is finished: for large projects maintaining software can keep many people busy for a long time.

Since there are so many factors influencing the success or failure of a project, there is a need to learn a little about project management and its pitfalls, but especially what makes projects successful. And last but not least, a good software engineer, like any engineer, needs tools, and to know about them is important.

Developers Work in Teams

In beginning coding was done by individuals. The problems solved earlier were small enough so one person could master them. In the real world this is different:- the problem sizes and time constraints are such that only teams can solve those problems.

For teams to work effectively they need a language to communicate (UML). Also teams do not consist only of developers, but also of testers, architects, system engineers and most

importantly the customer. There is a need to learn about what makes good teams, how to communicate with the customer, and how to document not only the source code, but everything related to the software project.

New Language

Programming languages, such as Java or C++, was used earlier and turn ideas into code. But these ideas are independent of the language. Unified Modeling Language (UML) is a way to describe code independently of language, and more importantly, it helps to think in one higher level of abstraction. UML can be an invaluable communication and documentation tool.

Pattern gives one higher level of abstraction. Again this increases our vocabulary to communicate more effectively with our peers. Also, it is a fantastic way to learn from our seniors. This is essential for designing large software systems.

Measurement

Also just being able to write software, doesn't mean that the software is any good. Discovering what makes good software, and how to measure software quality is necessary. Analysis of existing source code through static analysis and measuring metrics is needed.

It is needed to ensure that the code meets certain quality standards. Testing is also important in this context, it guarantees high quality products.

New Tools

Apart from an IDE, a compiler and a debugger, there are many more tools at the disposal of a software engineer. There are tools that allow us to work in teams, to document our software, to assist and monitor the whole development effort. There are tools for software architects, tools for testing and profiling, automation and re-engineering.

EVOLVING ROLE OF SOFTWARE

The industry originated with the entrepreneurial computer software and services companies of the 1950s and 1960s, grew dramatically through the 1970s and 1980s to become a market force rivaling that of the computer hardware companies, and by the 1990s had become the supplier of technical know-how that transformed the way people worked, played and communicated every day of their lives. The following are the different eras' of software engineering:

The Pioneering Era (1955-1965)

The most important development was that new computers were coming out almost every year or two, rendering existing ones obsolete. Software people had to rewrite all their programs to run on these new machines.

Jobs were run by signing up for machine time or by operational staff by putting punched cards for input into the machine's card reader and waiting for results to come back on the printer.

The field was so new that the idea of management by schedule was non-existent. Making predictions of a project's completion date was almost impossible.

Computer hardware was application-specific. Scientific and business tasks needed different machines.

Hardware vendors gave away systems software for free as hardware could not be sold without software. A few companies sold the service of building custom software but no software companies were selling packaged software.

The Stabilizing Era (1965-1980)

The whole job-queue system had been institutionalized and so programmers no longer ran their jobs except for peculiar applications like on-board computers. To handle the jobs, an enormous bureaucracy had grown up around the central computer center.

The major problem as a result of this bureaucracy was turnaround time, the time between job submission and completion. At worst it was measured in days.

Then came IBM 360. It signaled the beginning of the stabilizing era. This was the largest software project to date. The 360 also combined scientific and business applications onto one machine.

The job control language (JCL) raised a whole new class of problems. The programmer had to write the program in a whole new language to tell the computer and OS what to do. JCL was the least popular feature of the 360.

"Structured Programming" burst on the scene in the middle of this era. PL/I, introduced by IBM to merge all programming languages into one, failed. Most customized applications continued to be done in-house.

The Micro Era (1980-Present)

The price of computing has dropped dramatically making ubiquitous computing possible. Now every programmer can have a computer on his desk. The old JCL has been replaced by the user friendly GUI.

The software part of the hardware architecture that the programmer must know about, such as the instruction set, has not changed much since the advent of the IBM mainframe and the first Intel chip.

The most-used programming languages today are between 15 and 40 years old. The Fourth Generation Languages never achieved the dream of "programming without programmers" and the idea is pretty much limited to report generation from databases. There is an increasing clamor though for more and better software research.

Computer software continues to be the single most important technology on the world stage. And it’s also a prime example of the law of unintended consequences.

Evolution of software on Different Industries:

- Fifty years ago no one could have predicted that software would become an indispensable technology for business, science, and engineering; that software would enable the creation of new technologies (e.g., genetic engineering and nanotechnology), the extension of existing technologies (e.g., telecommunications), and the radical change in older technologies (e.g., the printing industry).
- Software would be the driving force behind the personal computer revolution; that shrink-wrapped software products would be purchased by consumers in neighborhood malls; that software would slowly evolve from a product to a service as "on-demand" software companies deliver just-in-time functionality via a Web browser;
- A software company would become larger and more influential than almost all industrial-era companies; that a vast software-driven network called the Internet would evolve and change everything from library research to consumer shopping to political discourse to the dating habits of young (and not so young) adults.
- As software's importance has grown, the software community has continually attempted to develop technologies that will make it easier, faster, and less expensive to build and maintain high-quality computer programs. Some of these technologies are targeted at a specific application domain (e.g., website design and implementation); others focus on a technology domain (e.g., object-oriented systems or aspect oriented programming); and still others are broad-based (e.g., operating systems such as Linux).
- However, we have yet to develop a software technology that does it all, and the likelihood of one arising in the future is small. And yet, people bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right.

SOFTWARE

Software is: (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

SOFTWARE CHARACTERISTICS

Software is developed or engineered; it is not manufactured in the classical sense.

Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software.

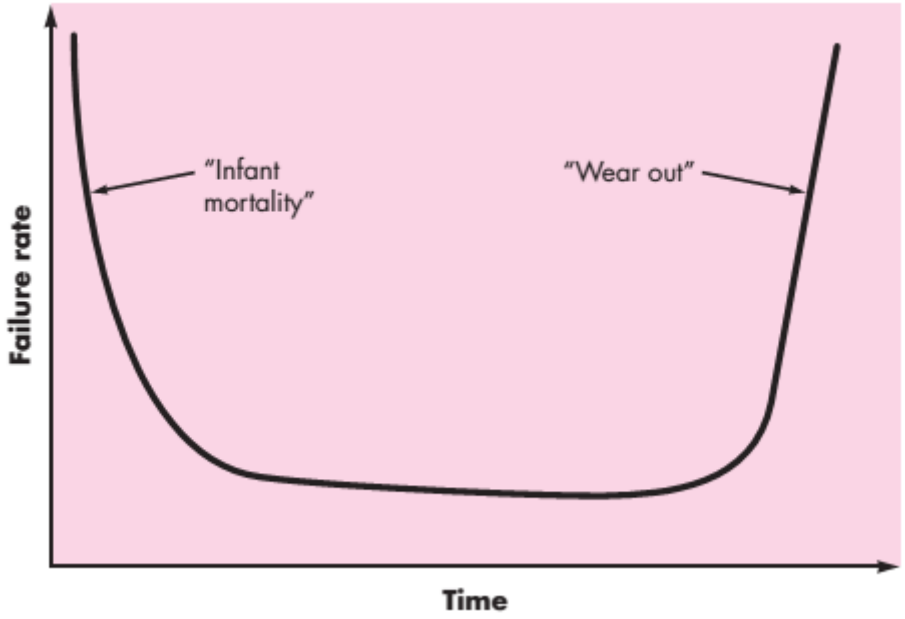


Fig 1.1. Failure curve for hardware

Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different. Both activities require the construction of a "product," but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

Software doesn't "wear out." Figure 1.1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to *wear out.*

Software is not susceptible to the environmental problems that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure 1.2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does *deteriorate.*

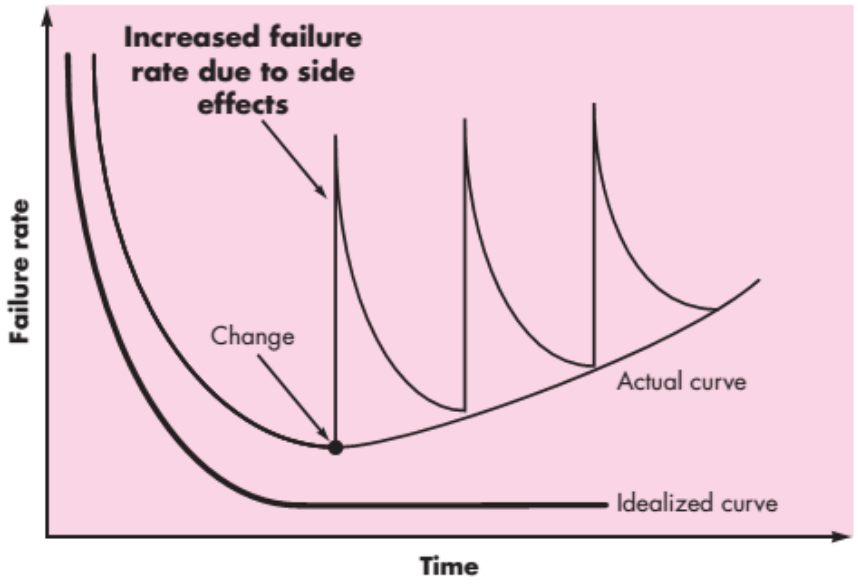


Fig 1.2. Failure curves for software

This seeming contradiction can best be explained by considering the actual curve in Figure 1.2. During its life, software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve" (Figure 1.2). Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts.

- Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.

Although the industry is moving toward component-based construction, most software continues to be custom built. As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new. In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

- **A software component** should be designed and implemented so that it can be reused in many different programs.
- Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new applications from reusable parts.
- For example, today's interactive user interfaces are built with reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

Software Application Domains

Today, seven broad categories of computer software present continuing challenges for software engineers:

System software—a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate,4 information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data. In either case, the systems software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

Application software—stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making. In addition to conventional data processing applications, application software is used to control business functions in real time (e.g., point-of-sale transaction processing, real-time manufacturing process control).

Engineering/scientific software—has been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

Embedded software—resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. Embedded software can perform limited and esoteric functions (e.g., key pad control for a microwave oven) or provide significant

function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

Product-line software—designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications).

Web applications—called "WebApps," this network-centric software category spans a wide array of applications. In their simplest form, WebApps can be little more than a set of linked hypertext files that present information using text and limited graphics. However, as Web 2.0 emerges, WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

Artificial intelligence software—makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

Millions of software engineers worldwide are hard at work on software projects in one or more of these categories. In some cases, new systems are being built, but in many others, existing applications are being corrected, adapted, and enhanced. It is not uncommon for a young software engineer to work a program that is older than she is! Past generations of software people have left a legacy in each of the categories I have discussed. Hopefully, the legacy to be left behind by this generation will ease the burden of future software engineers. And yet, new challenges (Chapter 31) have appeared on the horizon:

Open-world computing—the rapid growth of wireless networking may soon lead to true pervasive, distributed computing. The challenge for software engineers will be to develop systems and application software that will allow mobile devices, personal computers, and enterprise systems to communicate across vast networks

Netsourcing—the World Wide Web is rapidly becoming a computing engine as well as a content provider. The challenge for software engineers is to architect simple (e.g., personal financial planning) and sophisticated applications that provide a benefit to targeted end-user markets worldwide.

Open source—a growing trend that results in distribution of source code for systems applications (e.g., operating systems, database, and development environments) so that many

people can contribute to its development. The challenge for software engineers is to build source code that is self-descriptive, but more importantly, to develop techniques that will enable both customers and developers to know what changes have been made and how those changes manifest themselves within the software.

Each of these new challenges will undoubtedly obey the law of unintended consequences and have effects (for businesspeople, software engineers, and end users) that cannot be predicted today. However, software engineers can prepare by instantiating a process that is agile and adaptable enough to accommodate dramatic changes in technology and to business rules that are sure to come over the next decade.

Legacy Software

Hundreds of thousands of computer programs fall into one of the seven broad application domains discussed in the preceding subsection. Some of these are state of-the-art software—just released to individuals, industry, and government. But other programs are older, in some cases *much* older.

These older programs—often referred to as *legacy software*—have been the focus of continuous attention and concern since the 1960s.

Unfortunately, there is sometimes one additional characteristic that is present in legacy software—*poor quality*.

However, as time passes, legacy systems often evolve for one or more of the following reasons:

The software must be adapted to meet the needs of new computing environments or technology.

The software must be enhanced to implement new business requirements.

The software must be extended to make it interoperable with other more modern systems or databases.

The software must be re-architected to make it viable within a network environment.

When these modes of evolution occur, a legacy system must be reengineered so that it remains viable into the future. The goal of modern software engineering is to "devise methodologies that are founded on the notion of evolution"; that is, the notion that software systems continually change, new software systems are built from the old ones, and . . . all must interoperate and cooperate with each other.

SOFTWARE MYTHS

Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing.

Myths have a number of attributes that make them insidious. For instance, they appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often promulgated by experienced practitioners who "know the score."

Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike. However, old attitudes and habits are difficult to modify, and remnants of software myths remain.

Management myths.

Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

Myth: *We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?*

Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

Myth: *If we get behind schedule, we can add more programmers and catch up (sometimes called the "Mongolian horde" concept).*

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks [Bro95]: "adding people to a late software project makes it later." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well coordinated manner.

Myth: *If I decide to outsource the software project to a third party, I can just relax and let that firm build it.*

Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

Customer myths.

A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer.

Myth: *A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.*

Reality: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous

requirements (usually derive iteratively) are developed only through effective and continuous communication between customer and developer.

Myth: *Software requirements continually change, but change can be easily accommodated because software is flexible.*

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small.16 However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can cause upheaval that requires additional resources and major design modification.

Practitioner's myths.

Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

Myth: *Once we write the program and get it to work, our job is done.*

Reality: Someone once said that "the sooner you begin 'writing code,' the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: *Until I get the program "running" I have no way of assessing its quality.*

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—*the technical review.* Software reviews (described in Chapter 15) are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

Myth: *The only deliverable work product for a successful project is the working program.*

Reality: A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.

Myth: *Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.*

Reality: Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times

Many software professionals recognize the fallacy of the myths just described. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering.

A GENERIC VIEW OF PROCESS

A process is defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.

The software process is represented schematically in Figure 1.3. Referring to the figure, each framework activity is populated by a set of software engineering actions.

Each software engineering action is defined by a *task set* that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress.

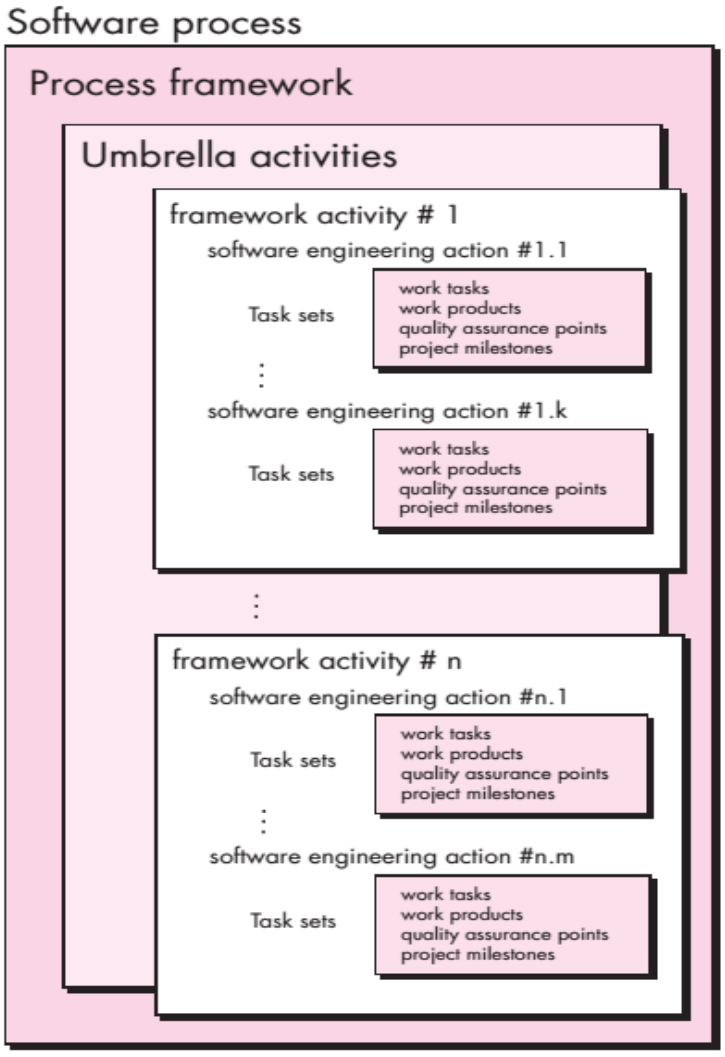


Fig 1.3. A software process framework

A generic process framework for software engineering defines five framework activities—**communication, planning, modeling, construction,** and **deployment.** In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.

One important aspect of the software process is called *process flow*—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in Figure 1.4.

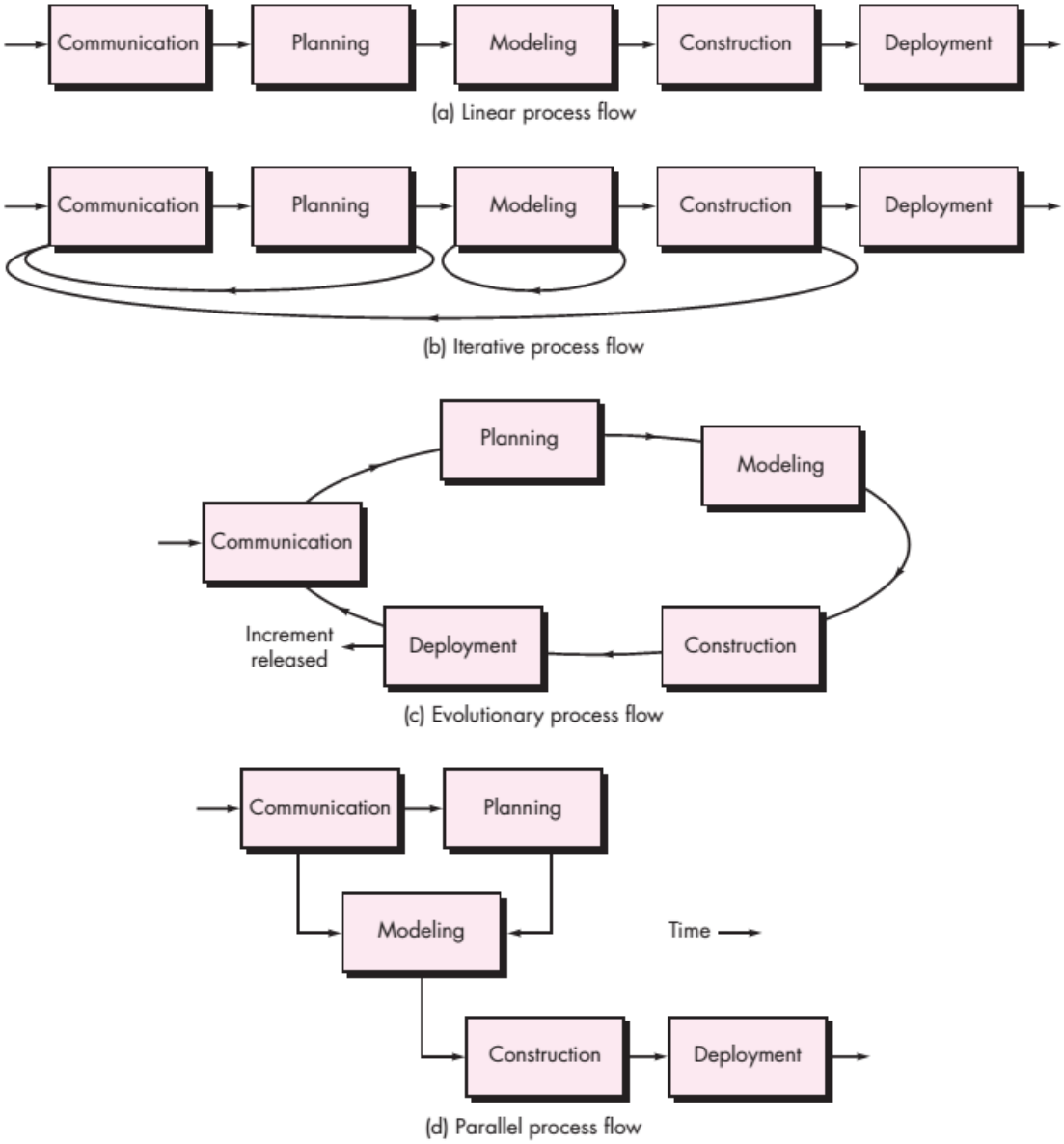


Fig 1.4. Process flow

- A *linear process flow* executes each of the five framework activities in sequence, beginning with communication and culminating with deployment (Figure 1.4a).

- An *iterative process flow* repeats one or more of the activities before proceeding to the next (Figure 1.4.b).
- An *evolutionary process flow* executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software (Figure 1.4c).
- A *parallel process flow* (Figure 1.4d) executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software)

Defining a Framework Activity

A software team would need significantly more information before it could properly execute any one of these activities as part of the software process.

A key question is: *What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project*

For a small software project requested by one person (at a remote location) with simple, straightforward requirements, the communication activity might encompass little more than a phone call with the appropriate stakeholder. Therefore, the only necessary action is *phone conversation,* and the work tasks (the *task set*) that this action encompasses are:

1. Make contact with stakeholder via telephone.

2. Discuss requirements and take notes.

3. Organize notes into a brief written statement of requirements.

4. E-mail to stakeholder for review and approval.

If the project was considerably more complex with many stakeholders, each with a different set of (sometime conflicting) requirements, the communication activity might have six distinct actions *inception, elicitation, elaboration, negotiation, specification,* and *validation.* Each of these software engineering actions would have many work tasks and a number of distinct work products.

Identifying a Task Set

Referring again to Figure 1.3 each software engineering action (e.g., *elicitation,* an action associated with the communication activity) can be represented by a

- Number of different *task sets*—each a collection of software engineering work tasks,
- Related work products,
- Quality assurance points,
- Project milestones.

Choose a task set that best accommodates the needs of the project and the characteristics of your team. This implies that a software engineering action can be adapted to the specific needs of the software project and the characteristics of the project team.

Task Set

A task set defines the actual work to be done to accomplish the objectives of a software engineering action. For example, *elicitation* (more commonly called "requirements gathering") is an important software engineering action that occurs during the communication activity. The goal of requirements gathering is to understand what various stakeholders want from the software that is to be built. For a small, relatively simple project, the task set for requirements gathering might look like this:

1. Make a list of stakeholders for the project.
2. Invite all stakeholders to an informal meeting.
3. Ask each stakeholder to make a list of features and functions required.
4. Discuss requirements and build a final list.
5. Prioritize requirements.
6. Note areas of uncertainty.

Process Patterns

Every software team encounters problems as it moves through the software process. It would be useful if proven solutions to these problems were readily available to the team so that the problems could be addressed and resolved quickly.

A *process pattern* describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem.

Stated in more general terms, a process pattern provides you with a template—a consistent method for describing problem solutions within the context of the software process. By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.

Patterns can be defined at any level of abstraction. In some cases, a pattern might be used to describe a problem (and solution) associated with a complete process model (e.g., prototyping). In other situations, patterns can be used to describe a problem (and solution) associated with a framework activity (e.g., **planning**) or an action within a framework activity (e.g., project estimating). Ambler has proposed a template for describing a process pattern:

Pattern Name. The pattern is given a meaningful name describing it within the context of the software process (e.g., **TechnicalReviews**).

Forces. The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

Type. The pattern type is specified. Ambler [Amb98] suggests three types:

1. *Stage pattern*—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity). An example of a stage pattern might be **EstablishingCommunication.** This pattern would incorporate the task pattern **RequirementsGathering** and others.

2. *Task pattern*—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., **RequirementsGathering** is a task pattern).

3. *Phase pattern*—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature. An example of a phase pattern might be **SpiralModel** or **Prototyping.**

Initial context. Describes the conditions under which the pattern applies. Prior to the initiation of the pattern: (1) What organizational or team-related activities have already occurred? (2) What is the entry state for the process? (3) What software engineering information or project information already exists? For example, the **Planning** pattern (a stage pattern) requires that (1) customers and software engineers have established a collaborative communication; (2) successful completion of a number of task patterns [specified] for the

Communication pattern has occurred; and (3) the project scope, basic business requirements, and project constraints are known.

Problem. The specific problem to be solved by the pattern.

Solution. Describes how to implement the pattern successfully. This section describes how the initial state of the process (that exists before the pattern is implemented) is modified as a consequence of the initiation of the pattern. It also describes how software engineering information or project information that is available before the initiation of the pattern is transformed as a consequence of the successful execution of the pattern.

Resulting Context. Describes the conditions that will result once the pattern has been successfully implemented. Upon completion of the pattern: (1) What organizational or team-related activities must have occurred? (2) What is the exit state for the process? (3) What software engineering information or project information has been developed?

Related Patterns. Provide a list of all process patterns that are directly related to this one. This may be represented as a hierarchy or in some other diagrammatic form. For example, the stage pattern **Communication** encompasses the task patterns: ProjectTeam, CollaborativeGuidelines, ScopeIsolation, RequirementsGathering, ConstraintDescription, and ScenarioCreation**.**

Known Uses and Examples. Indicate the specific instances in which the pattern is applicable. For example, **Communication** is mandatory at the beginning of every software project, is recommended throughout the software project, and is mandatory once the deployment activity is under way.

Process patterns provide an effective mechanism for addressing problems associated with any software process. The patterns enable you to develop a hierarchical process description that begins at a high level of abstraction (a phase pattern).

The description is then refined into a set of stage patterns that describe framework activities and are further refined in a hierarchical fashion into more detailed task patterns for each stage pattern. Once process patterns have been developed, they can be reused for the definition of process variants—that is, a customized process model can be defined by a software team using the patterns as building blocks for the process model.

SOFTWARE ENGINEERING

In order to build software that is ready to meet the challenges of the twenty-first century, you must recognize a few simple realities:

- ***It follows that a concerted effort should be made to understand the problem before a software solution is developed.***

• Software has become deeply embedded in virtually every aspect of our lives, and as a consequence, the number of people who have an interest in the features and functions provided by a specific application has grown dramatically. When a new application or embedded system is to be built, many voices must be heard. And it sometimes seems that each of them has a slightly different idea of what software features and functions should be delivered.

- ***It follows that design becomes a pivotal activity.***

• The information technology requirements demanded by individuals, businesses, and governments grow increasing complex with each passing year. Large teams of people now create computer programs that were once built by a single individual. Sophisticated software that was once implemented in a predictable, self-contained, computing environment is now embedded inside everything from consumer electronics to medical devices to weapons systems. The complexity of these new computer-based systems and products demands careful attention to the interactions of all system elements.

- ***It follows that software should exhibit high quality.***

• Individuals, businesses, and governments increasingly rely on software for strategic and tactical decision making as well as day-to-day operations and control. If the software fails, people and major enterprises can experience anything from minor inconvenience to catastrophic failures.

- ***It follows that software should be maintainable.***

• As the perceived value of a specific application grows, the likelihood is that its user base and longevity will also grow. As its user base and time-in-use increase, demands for adaptation and enhancement will also grow.

These simple realities lead to one conclusion: *software in all of its forms and across all of its application domains should be engineered.* And that leads us to the topic of this book—*software engineering.*

Although hundreds of authors have developed personal definitions of software engineering, a definition proposed by Fritz Bauer [Nau69] at the seminal conference on the subject still serves as a basis for discussion:

- **[Software engineering is]** the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.
- **Software Engineering:** (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1). And yet, a "systematic, disciplined, and quantifiable" approach applied by one software team may be burdensome to another. We need discipline, but we also need adaptability and agility.

A LAYERED TECHNOLOGY

Software engineering is a layered technology. Referring to Figure 1.5, any engineering approach (including software engineering) must rest on an organizational commitment to quality.

Total quality management, Six Sigma, and similar philosophies10 foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering.

The bedrock that supports software engineering is a quality focus. The foundation for software engineering is the *process* layer.

The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software.

Fig 1.5 Software engineering layers

Process defines a framework that must be established for effective delivery of software engineering technology.

The software process forms the basis for

- Management control of software projects and establishes the context in which technical methods are applied
- Work products (models, documents, data, reports, forms, etc.) are produced

- Milestones are established
- Quality is ensured
- Change is properly managed.

Software engineering *methods* provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Software engineering *tools* provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called *computer-aided software engineering*, is established.

PROCESS MODELS:

PRESCRIPTIVE MODELS

Prescriptive process models were originally proposed to bring order to the chaos of software development. History has indicated that these traditional models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams. However, software engineering work and the product that it produces remain on "the edge of chaos."

Change occurs when there is some structure so that the change can be organized, but not so rigid that it cannot occur. Too much chaos, on the other hand, can make coordination and coherence impossible. Lack of structure does not always mean disorder. The philosophical implications of this argument are significant for software engineering.

If prescriptive process models strive for structure and order, are they inappropriate for a software world that thrives on change? Yet, if we reject traditional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

There are no easy answers to these questions, but there are alternatives available to software engineers. In the sections that follow, I examine the prescriptive process approach in which order and project consistency are dominant issues. I call them "prescriptive" because they prescribe a set of process elements—framework activities, software engineering actions, tasks, work products, quality assurance, and change control mechanisms for each project. Each process model also prescribes a process flow (also called a *work flow*)—that is, the manner in which the process elements are interrelated to one another.

WATERFALL MODEL

There are times when the requirements for a problem are well understood—when work flows from **communication** through **deployment** in a reasonably linear fashion. This situation is sometimes encountered when well-defined adaptations or enhancements to an existing system must be made (e.g., an adaptation to accounting software that has been mandated because of

changes to government regulations). It may also occur in a limited number of new development efforts, but only when requirements are well defined and reasonably stable.

The *waterfall model*, sometimes called the *classic life cycle*, suggests a systematic, sequential approach6 to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software (Figure 1.6).

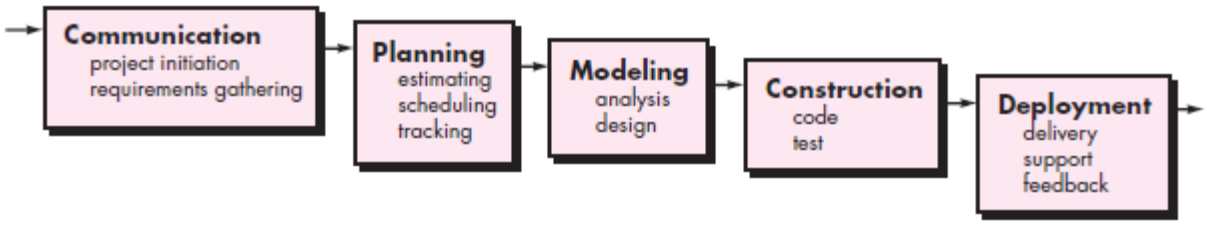


Fig 1.6.The waterfall model

The waterfall model is the oldest paradigm for software engineering. However, over the past three decades, criticism of this process model has caused even passionate supporters to question its efficacy. Among the problems that are sometimes encountered when the waterfall model is applied are:

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

3. The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

In an interesting analysis of actual projects, Bradac found that the linear nature of the classic life cycle leads to "blocking states" in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent waiting can exceed the time spent on productive work! The blocking states tend to be more prevalent at the beginning and end of a linear sequential process.

Today, software work is fast-paced and subject to a never-ending stream of changes (to features, functions, and information content). The waterfall model is often inappropriate for such work. However, it can serve as a useful process model in situations where requirements are fixed and work is to proceed to completion in a linear manner.

INCREMENTAL PROCESS MODELS

There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process. In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases. In such cases, you can choose a process model that is designed to produce the software in increments.

The *incremental* model combines elements of linear and parallel process. Referring to Figure 1.7, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software in a manner that is similar to the increments produced by an evolutionary process flow.

For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment. It should be noted that the process flow for any increment can incorporate the prototyping paradigm.

When an incremental model is used, the first increment is often a *core product.* That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation).

As a result of use and/or evaluation, a plan is developed for the next increment.

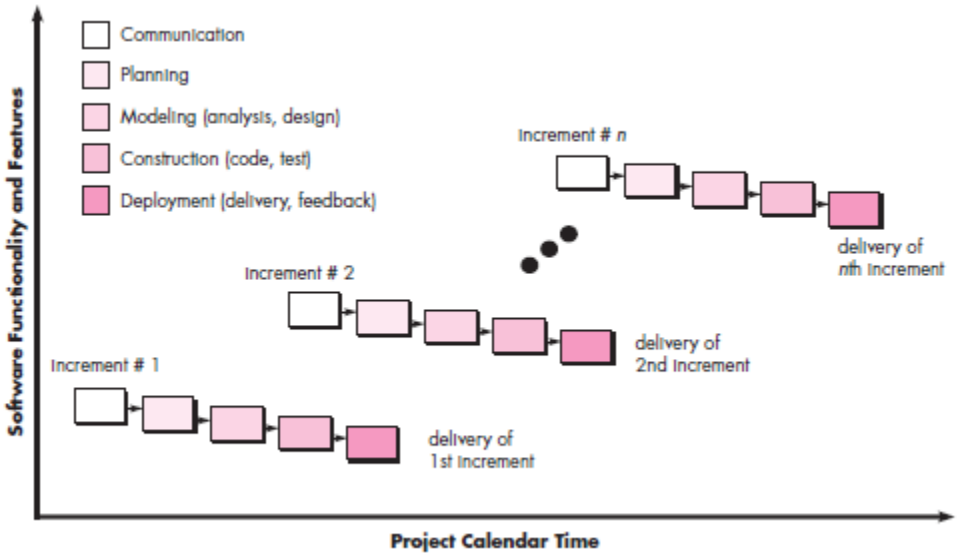


Fig 1.7. The incremental model

The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment.

In addition, increments can be planned to manage technical risks. For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end users without inordinate delay.

EVOLUTIONARY PROCESS MODELS

Software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight line path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined.

In these and similar situations, a process model that has been explicitly designed to accommodate a product that evolves over time is needed. Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software.

PROTOTYPING

Prototyping. Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a *prototyping paradigm* may offer the best approach.

Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models. Regardless of the manner in which it is applied, the prototyping paradigm assists you and other stakeholders to better understand what is to be built when requirements are fuzzy.

The prototyping paradigm (Figure 2.6) begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory. A prototyping iteration is planned quickly, and modeling (in the form of a "quick design") occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats).

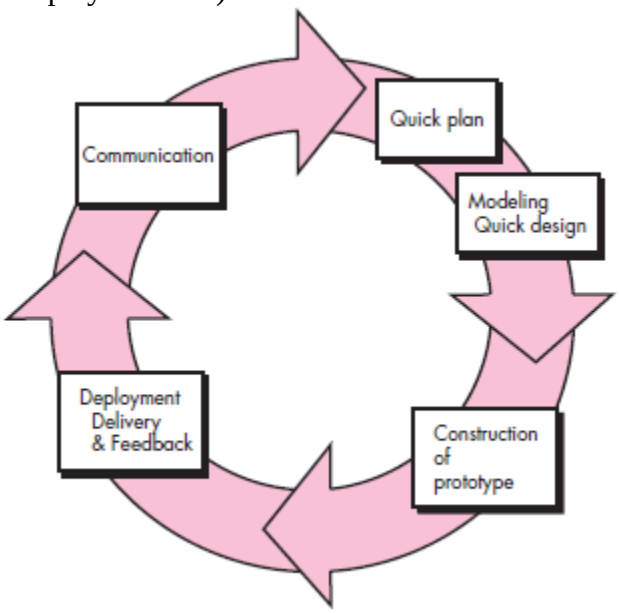


Fig 1.8. The prototyping paradigm

The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built, you can make use of existing program fragments or apply tools (e.g., report generators and window managers) that enable working programs to be generated quickly.

In most projects, the first system built is barely usable. It may be too slow, too big, awkward in use or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved.

The prototype can serve as "the first system." Although some prototypes are built as "throwaways," others are evolutionary in the sense that the prototype slowly evolves into the actual system.

Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately. Yet, prototyping can be problematic for the following reasons:

1. Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, stakeholders cry foul and demand that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.

2. As a software engineer, you often make implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

Although problems can occur, prototyping can be an effective paradigm for software engineering. The key is to define the rules of the game at the beginning; that is, all stakeholders should agree that the prototype is built to serve as a mechanism for defining requirements. It is then discarded (at least in part), and the actual software is engineered with an eye toward quality.

THE SPIRAL MODEL

The Spiral Model. Originally proposed by Barry Boehm, the *spiral model* is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provides the potential for rapid development of increasingly more complete versions of the software. Boehm describes the model in the following manner:

The spiral development model is a *risk*-driven *process model* generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a *cyclic* approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of *anchor point milestones* for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

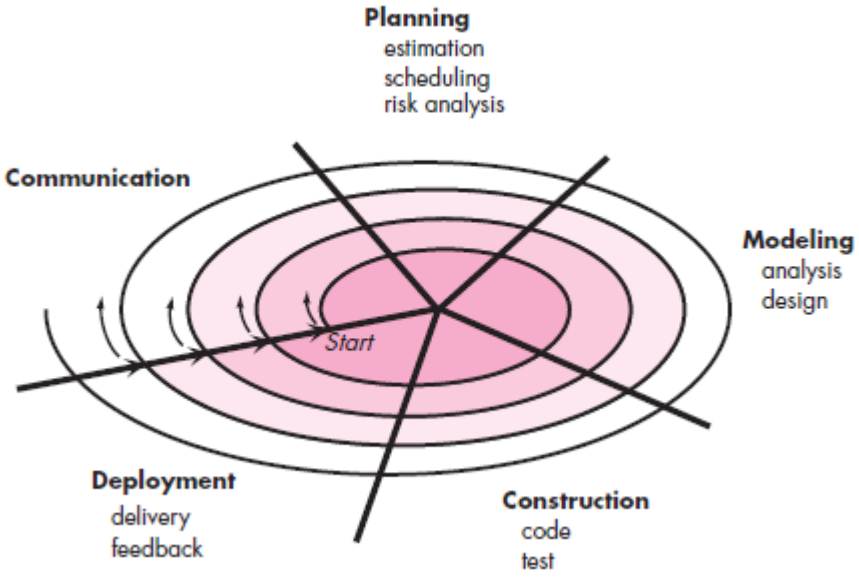


Fig 1.9. A typical spiral model

A spiral model is divided into a set of framework activities defined by the software engineering team. For illustrative purposes, I use the generic framework activities discussed earlier. Each of the framework activities represent one segment of the spiral path illustrated in Figure 1.9. As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center. Risk is considered as each revolution is made. *Anchor point milestones*—a combination of work products and conditions that are attained along the path of the spiral—are noted for each evolutionary pass.

The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.

Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software. Therefore, the first circuit around the spiral might represent a "concept development project" that starts at the core of the spiral and continues for multiple iterations until concept development is complete.

If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "new product development project" commences. The new product will evolve through a number of iterations around the spiral. Later, a circuit around the spiral might be used to represent a "product enhancement project." In essence, the spiral, when characterized in this way, remains operative until the software is retired. There are times when the process is dormant,

but whenever a change is initiated, the process starts at the appropriate entry point (e.g., product enhancement).

The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level.

The spiral model uses prototyping as a risk reduction mechanism but, more important, enables you to apply the prototyping approach at any stage in the evolution of the product. It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world. The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.

But like other paradigms, the spiral model is not a panacea. It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur.

A Final Word on Evolutionary Processes

Modern computer software is characterized by continual change, by very tight time lines, and by an emphatic need for customer–user satisfaction. In many cases, time-to-market is the most important management requirement. If a market window is missed, the software project itself may be meaningless.

Evolutionary process models were conceived to address these issues, and yet, as a general class of process models, they too have weaknesses. Despite the unquestionable benefits of evolutionary software processes, we have some concerns. The first concern is that prototyping [and other more sophisticated evolutionary processes] poses a problem to project planning because of the uncertain number of cycles required to construct the product. Most project management and estimation techniques are based on linear layouts of activities, so they do not fit completely.

Second, evolutionary software processes do not establish the maximum speed of the evolution. If the evolutions occur too fast, without a period of relaxation, it is certain that the process will fall into chaos. On the other hand if the speed is too slow then productivity could be affected.

Third, software processes should be focused on flexibility and extensibility rather than on high quality. This assertion sounds scary. However, we should prioritize the speed of the development over zero defects. Extending the development in order to reach high quality could result in a late delivery of the product, when the opportunity niche has disappeared. This paradigm shift is imposed by the competition on the edge of chaos.

Indeed, a software process that focuses on flexibility, extensibility, and speed of development over high quality does sound scary. And yet, this idea has been proposed by a number of well-respected software engineering experts.

The intent of evolutionary models is to develop high-quality software14 in an iterative or incremental manner. However, it is possible to use an evolutionary process to emphasize

flexibility, extensibility, and speed of development. The challenge for software teams and their managers is to establish a proper balance between these critical project and product parameters and customer satisfaction (the ultimate arbiter of software quality).

SPECIALIZED PROCESS MODELS

Specialized process models take on many of the characteristics of one or more of the traditional models presented in the preceding sections. However, these models tend to be applied when a specialized or narrowly defined software engineering approach is chosen.

Component-Based Development

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built. The *component-based development model* incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from prepackaged software components.

Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

1. Available component-based products are researched and evaluated for the application domain in question.

2. Component integration issues are considered.

3. A software architecture is designed to accommodate the components.

4. Components are integrated into the architecture.

5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Your software engineering team can achieve a reduction in development cycle time as well as a reduction in project cost if component reuse becomes part of your culture

The Formal Methods Model

The *formal methods model* encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A

variation on this approach, called *cleanroom software engineering*, is currently applied by some software development organizations.

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily—not through ad hoc review, but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable you to discover and correct errors that might otherwise go undetected.

Although not a mainstream approach, the formal methods model offers the promise of defect-free software. Yet, concern about its applicability in a business environment has been voiced:

• The development of formal models is currently quite time consuming and expensive.

• Because few software developers have the necessary background to apply formal methods, extensive training is required.

• It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

These concerns notwithstanding, the formal methods approach has gained adherents among software developers who must build safety-critical software (e.g., developers of aircraft avionics and medical devices) and among developers that would suffer severe economic hardship should software errors occur.

Aspect-Oriented Software Development

Regardless of the software process that is chosen, the builders of complex software invariably implement a set of localized features, functions, and information content. These localized software characteristics are modeled as components (e.g., objectoriented classes) and then constructed within the context of a system architecture.

As modern computer-based systems become more sophisticated (and complex), certain *concerns*—customer required properties or areas of technical interest—span the entire architecture. Some concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect functions (e.g., the application of business rules), while others are systemic (e.g., task synchronization or memory management).

When concerns cut across multiple system functions, features, and information, they are often referred to as *crosscutting concerns. Aspectual requirements* define those crosscutting concerns that have an impact across the software architecture.

Aspect-oriented software development (AOSD), often referred to as *aspect-oriented programming* (AOP), is a relatively new software engineering paradigm that provides a process

and methodological approach for defining, specifying, designing, and constructing *aspects*—"mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern".

Aspect-oriented component engineering (AOCE):

AOCE uses a concept of horizontal slices through vertically-decomposed software components, called "aspects," to characterize cross-cutting functional and non-functional properties of components. Common, systemic aspects include user interfaces, collaborative work, distribution, persistency, memory management, transaction processing, security, integrity and so on.

Components may provide or require one or more "aspect details" relating to a particular aspect, such as a viewing mechanism, extensible affordance and interface kind (user interface aspects); event generation, transport and receiving (distribution aspects); data store/retrieve and indexing (persistency aspects); authentication, encoding and access rights (security aspects); transaction atomicity, concurrency control and logging strategy (transaction aspects); and so on. Each aspect detail has a number of properties, relating to functional and/or non-functional characteristics of the aspect detail.

A distinct aspect-oriented process has not yet matured. However, it is likely that such a process will adopt characteristics of both evolutionary and concurrent process models. The evolutionary model is appropriate as aspects are identified and then constructed. The parallel nature of concurrent development is essential because aspects are engineered independently of localized software components and yet, aspects have a direct impact on these components. Hence, it is essential to instantiate asynchronous communication between the software process activities applied to the engineering and construction of aspects and components.

POSSIBLE QUESTIONS
PART – B

1. Explain the different phases involved in waterfall life cycle. Give the reasons for the Failure of Water Fall Model.
2. Discuss on various types of software myths and the true aspects of the myths.
3. Explain about the Generic view of process in detail.
4. Elucidate the process model that combines the elements of waterfall and iterative fashion.
5. Explain the process model which is useful when staffing is unavailable to complete implementation.
6. Explain about the Evolutionary Process Model
7. Describe the Prescriptive process model in detail.
8. Explain with diagram the layered technology of software process along with its characteristics.
9. Explicate how the specialized models applied for software engineering approaches.

UNIT-II

SYLLABUS

Building the Analysis Model: Requirements Analysis-Analysis Modeling Approaches-Data Modeling Concepts: Data Objects-Date attributes-Relationships Cardinality and Modality-Flow Oriented Modeling: Creating Data Flow Model-Creating a Control Flow Model-The Control Specification-The Process Specification- Creating a Behavioral Model.

BUILDING THE ANALYSIS MODEL:

At a technical level, software engineering begins with a series of modeling tasks that lead to a specification of requirements and a design representation for the software to be built. The requirements model— actually a set of models—is the first technical representation of a system.

REQUIREMENTS ANALYSIS

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows you (regardless of whether you're called a *software engineer*, an *analyst*, or a *modeler*) to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering.

The requirements modeling action results in one or more of the following types of models:

• *Scenario-based models* of requirements from the point of view of various system "actors"

• *Data models* that depict the information domain for the problem

• *Class-oriented models* that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements

• *Flow-oriented models* that represent the functional elements of the system and how they transform data as it moves through the system

• *Behavioral models* that depict how the software behaves as a consequence of external "events"

These models provide a software designer with information that can be translated to architectural, interface, and component-level designs. Finally, the requirements model (and the

software requirements specification) provides the developer and the customer with the means to assess quality once software is built.

Scenario-based modeling—a technique that is growing increasingly popular throughout the software engineering community; *data modeling*—a more specialized technique that is particularly appropriate when an application must create or manipulate a complex information space; and *class modeling*—a representation of the object-oriented classes and the resultant collaborations that allow a system to function.

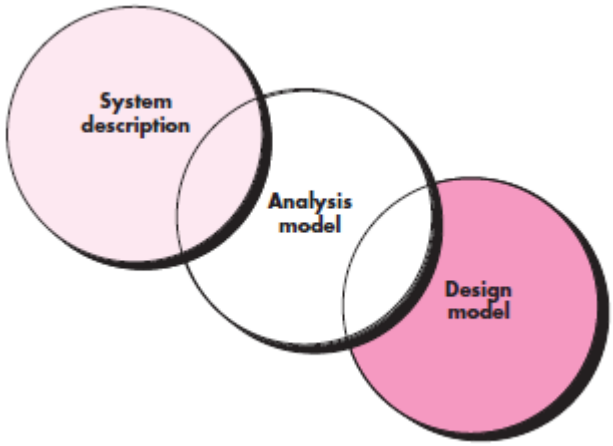


Fig 2.1. The requirements model as a bridge between the system description and the design model

Overall Objectives and Philosophy

Throughout requirements modeling, your primary focus is on *what*, not *how*. What user interaction occurs in a particular circumstance, what objects does the system manipulate, what functions must the system perform, what behaviors does the system exhibit, what interfaces are defined, and what constraints apply?

The customer may be unsure of precisely what is required for certain aspects of the system. The developer may be unsure that a specific approach will properly accomplish function and performance. These realities mitigate in favor of an iterative approach to requirements analysis and modeling. The analyst should model what is known and use that model as the basis for design of the software increment.

The requirements model must achieve three primary objectives:

(1) to describe what the customer requires,

(2) to establish a basis for the creation of a software design, and

(3) to define a set of requirements that can be validated once the software is built.

The analysis model bridges the gap between a system-level description that describes overall system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design that describes the software's application architecture, user interface, and component-level structure. This relationship is illustrated in Figure 2.1.

It is important to note that all elements of the requirements model will be directly traceable to parts of the design model. A clear division of analysis and design tasks between these two important modeling activities is not always possible. Some design invariably occurs as part of analysis, and some analysis will be conducted during design.

Analysis Rules of Thumb

Arlow and Neustadt suggest a number of worthwhile rules of thumb that should be followed when creating the analysis model:

- *The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.* "Don't get bogged down in details" that try to explain how the system will work.
- *Each element of the requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system.*
- *Delay consideration of infrastructure and other nonfunctional models until design.* That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.
- *Minimize coupling throughout the system.* It is important to represent relationships between classes and functions. However, if the level of "interconnectedness" is extremely high, effort should be made to reduce it.

- *Be certain that the requirements model provides value to all stakeholders.* Each constituency has its own use for the model. For example, business stakeholders should use the model to validate requirements; designers should use the model as a basis for design; QA people should use the model to help plan acceptance tests.

- *Keep the model as simple as it can be.* Don't create additional diagrams when they add no new information. Don't use complex notational forms, when a simple list will do.

Domain Analysis

In the discussion of requirements engineering, I noted that analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows you to recognize and apply them to solve common problems, the creation of the analysis model is expedited. More important, the likelihood of applying design patterns and executable software components grows dramatically. This improves time-to-market and reduces development costs.

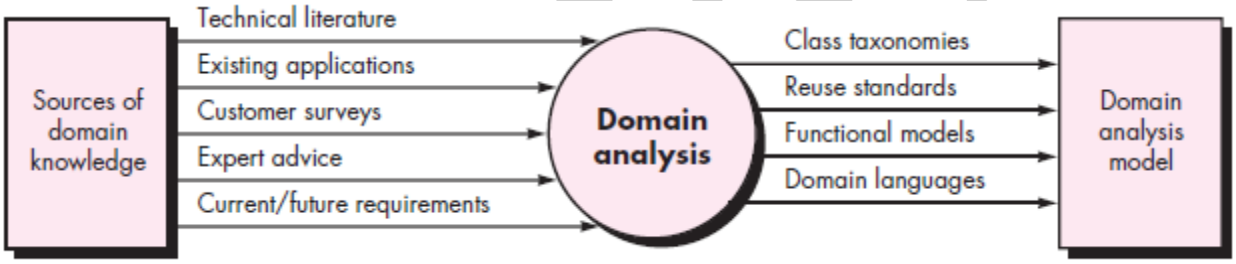


Fig 2.2. Input and output for domain analysis

But how are analysis patterns and classes recognized in the first place? Who defines them, categorizes them, and readies them for use on subsequent projects? The answers to these questions lie in *domain analysis.* Firesmith describes domain analysis in the following way:

Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain. . . . [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks.

The "specific application domain" can range from avionics to banking, from multimedia video games to software embedded within medical devices. The goal of domain analysis is straightforward: to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.

Using terminology that was introduced earlier in this book, domain analysis may be viewed as an umbrella activity for the software process. By this I mean that domain analysis is an ongoing software engineering activity that is not connected to any one software project. In a way, the role of a domain analyst is similar to the role of a master toolsmith in a heavy manufacturing environment. The job of the toolsmith is to design and build tools that may be used by many people doing similar but not necessarily the same jobs.

The role of the domain analyst5 is to discover and define analysis patterns, analysis classes, and related information that may be used by many people working on similar but not necessarily the same applications. Figure 6.2 [Ara89] illustrates key inputs and outputs for the domain analysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain.

ANALYSIS MODELING APPROACHES

Requirements Modeling Approaches

One view of requirements modeling, called *structured analysis*, considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.

A second approach to analysis modeling, called *object-oriented analysis*, focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process are predominantly object oriented.

Although the requirements model proposed in this book combines features of both approaches, software teams often choose one approach and exclude all representations from the other.

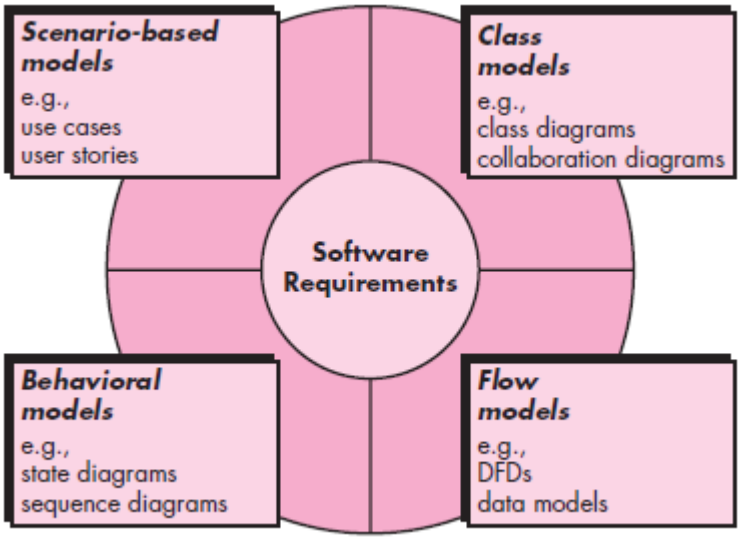


Fig 2.3. Elements of the analysis model

The question is not which is best, but rather, what combination of representations will provide stakeholders with the best model of software requirements and the most effective bridge to software design. Each element of the requirements model (Figure 2.3) presents the problem from a different point of view. Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used.

Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. Behavioral elements depict how external events change the state of the system or the classes that reside within it. Finally, flow-oriented elements represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions.

Analysis modeling leads to the derivation of each of these modeling elements. However, the specific content of each element (i.e., the diagrams that are used to construct the element and the model) may differ from project to project. As we have noted a number of times in this book, the software team must work to keep it simple. Only those modeling elements that add value to the model should be used.

DATA MODELING CONCEPTS:

If software requirements include the need to create, extend, or interface with a database or if complex data structures must be constructed and manipulated, the software team may choose to create a *data model* as part of overall requirements modeling. A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships. The *entity-relationship diagram* (ERD) addresses these issues and represents all data objects that are entered, stored, transformed, and produced within an application.

Data Objects

A *data object* is a representation of composite information that must be understood by software. By *composite information*, I mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but **dimensions** (incorporating height, width, and depth) could be defined as an object.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a **person** or a **car** can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The description of the data object incorporates the data object and all of its attributes.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data. Therefore, the data object can be represented as a table as shown in Figure 6.7. The headings in the table reflect attributes of the object. In this case, a car is defined in terms of make, model, ID number, body type, color, and owner. The body of the table represents specific instances of the data object. For example, a Chevy Corvette is an instance of the data object car.

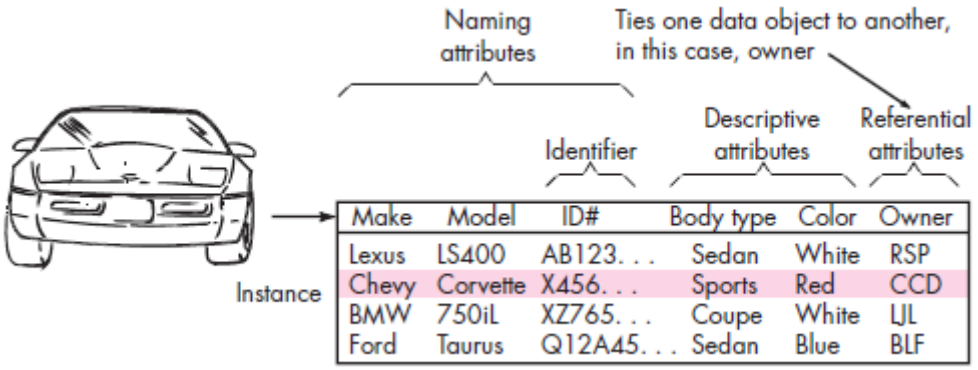
DATA ATTRIBUTES

Data attributes define the properties of a data object and take on one of three different characteristics. They can be used to

(1) name an instance of the data object,

(2) describe the instance, or

(3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier—that is, the identifier



Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123. . .	Sedan	White	RSP
Chevy	Corvette	X456. . .	Sports	Red	CCD
BMW	750iL	XZ765. . .	Coupe	White	LJL
Ford	Taurus	Q12A45. . .	Sedan	Blue	BLF

Fig 6.7 Tabular representation of data objects

attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object **car,** a reasonable identifier might be the ID number.

The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context. The attributes for **car** might serve well for an application that would be used by a department of motor vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software. In the latter case, the attributes for **car** might also include ID number, body type, and color, but many additional attributes (e.g., interior code, drive train type, trim package designator, transmission type) would have to be added to make **car** a meaningful object in the manufacturing control context.

Data Objects, Attributes, and Relationships

The data model consists of three interrelated pieces of information: the data object, the attributes that describe the data object, and the relationships that connect data objects to one another.

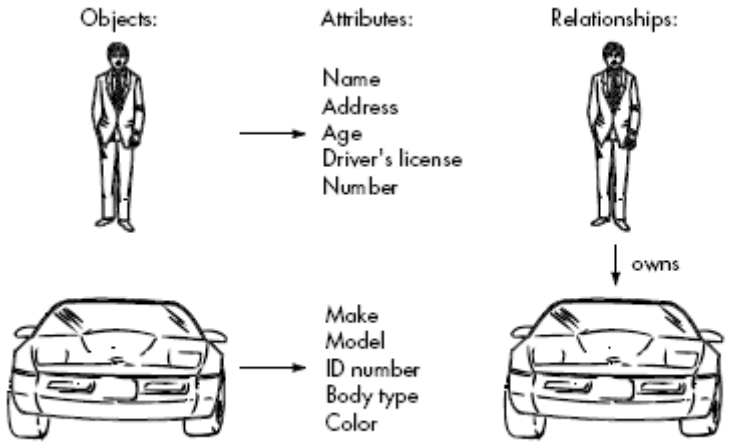


FIGURE 12.2 Data objects, attributes and relationships

Relationships. Data objects are connected to one another in different ways. Consider two data objects, **book** and **bookstore**. These objects can be represented using the simple notation illustrated in Figure 12.4a. A connection is established between **book** and **bookstore** because the two objects are related. But what are the relationships? To determine the answer, we must understand the role of books and bookstores within the context of the software to be built. We can define a set of object/relationship pairs that define the relevant relationships. For example,

A bookstore orders books.

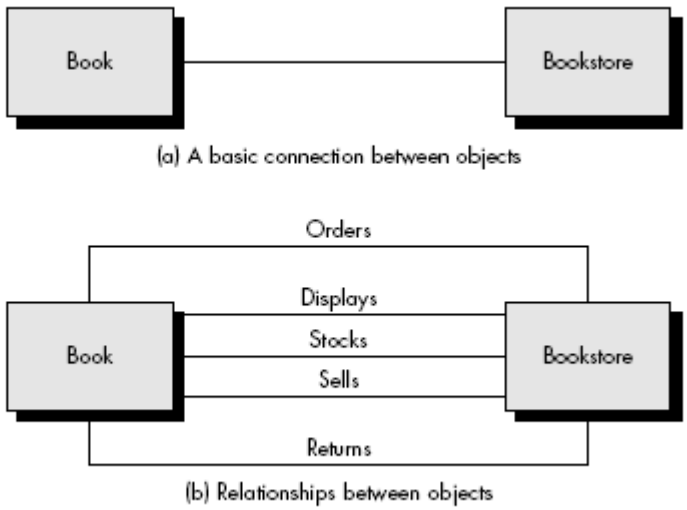
• A bookstore displays books.

• A bookstore stocks books.

• A bookstore sells books.

• A bookstore returns books.

FIGURE 12.4
Relationships



The relationships orders, displays, stocks, sells, and returns define the relevant connections between book and bookstore. Figure 12.4b illustrates these object/relationship pairs graphically. It is important to note that object/relationship pairs are bidirectional. That is, they can be read in either direction. A bookstore orders books or books are ordered by a bookstore.

Cardinality and Modality

• Cardinality is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object].

• Cardinality is usually expressed as simply 'one' or 'many.'

• Cardinality defines "the maximum number of objects that can participate in a relationship".

• It does not, however, provide an indication of whether or not a particular data object must participate in the relationship. To specify this information, the data model adds modality to the object/relationship pair

Modality

• The modality of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional.

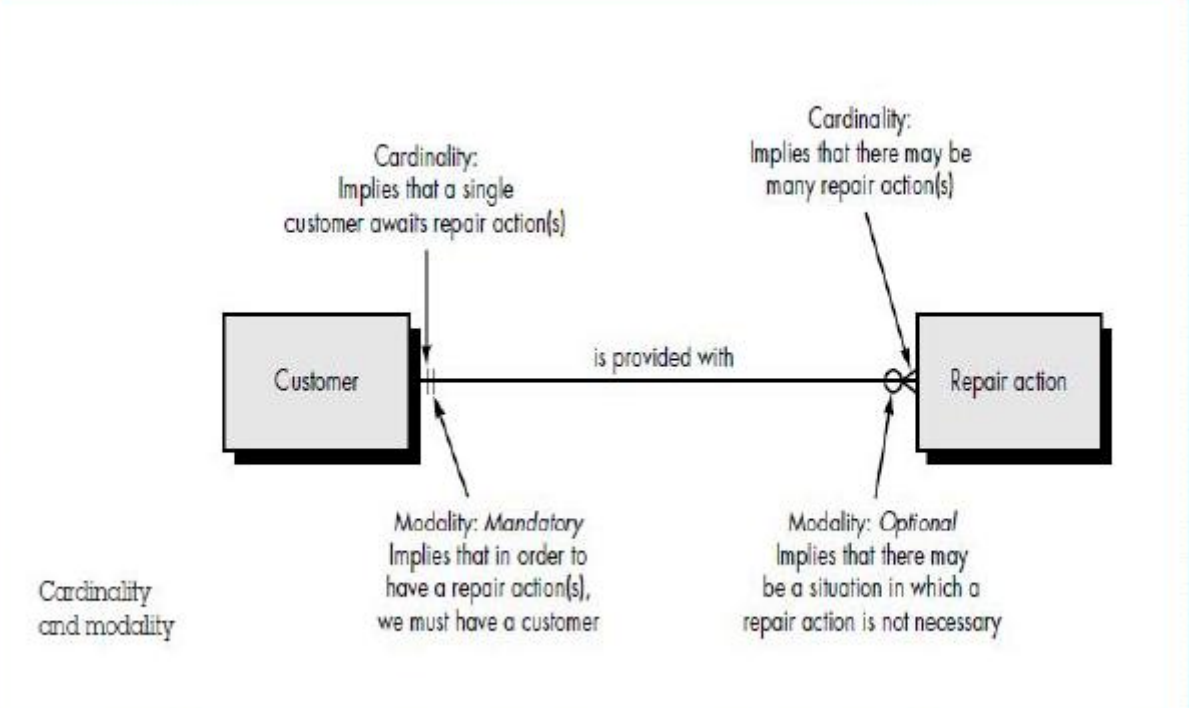
• The modality is 1 if an occurrence of the relationship is mandatory.

Example

• Consider software that is used by a local telephone company to process requests for field service. A customer indicates that there is a problem. If the problem is diagnosed as relatively

simple, a single repair action occurs. However, if the problem is complex, multiple repair actions may be required.

• Following figure illustrates the relationship, cardinality, and modality between the data objects customer and repair action.



Cardinality and modality

Entity/Relationship Diagrams

The object/relationship pair is the cornerstone of the data model. These pairs can be represented graphically using the *entity/relationship diagram*. The ERD was originally proposed by Peter Chen for the design of relational database systems and has been extended by others. A set of primary components are identified for the ERD: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships.

Data objects are represented by a labeled rectangle. Relationships are indicated with a labeled line connecting objects. In some variations of the ERD, the connecting line contains a diamond that is labeled with the relationship. Connections between data objects and relationships are established using a variety of special symbols that indicate cardinality and modality. The relationship between the data objects **car** and **manufacturer** would be represented as shown in Figure 12.6. One manufacturer builds one or many cars. Given the context implied by the ERD, the specification of the data object **car** (data object table in Figure 12.6) would be radically different from the earlier specification (Figure 12.3). By examining the symbols at the end of the

connection line between objects, it can be seen that the modality of both occurrences is mandatory (the vertical lines).

Expanding the model, we represent a grossly oversimplified ERD (Figure 12.7) of the distribution element of the automobile business. New data objects, **shipper** and **dealership,** are introduced. In addition, new relationships—*transports, contracts, licenses,* and *stocks*—indicate how the data objects shown in the figure associate with one another

FIGURE 12.6
A simple ERD and data object table (Note: In this ERD the relationship *builds* is indicated by a diamond)

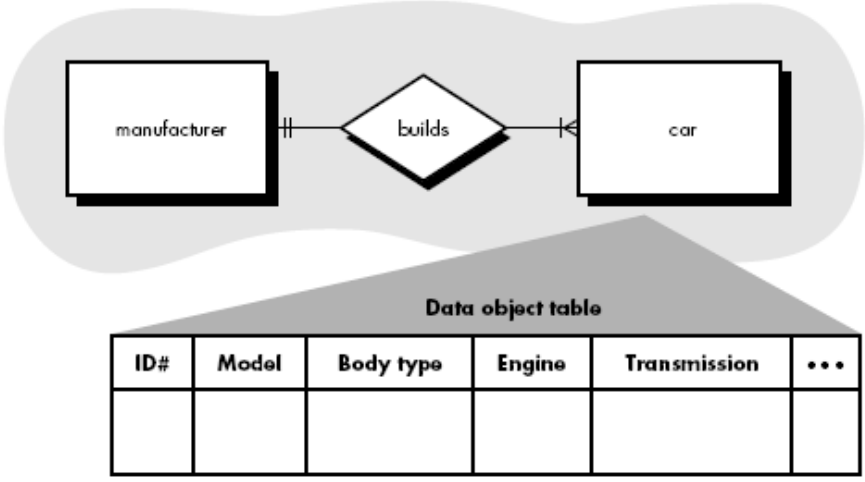
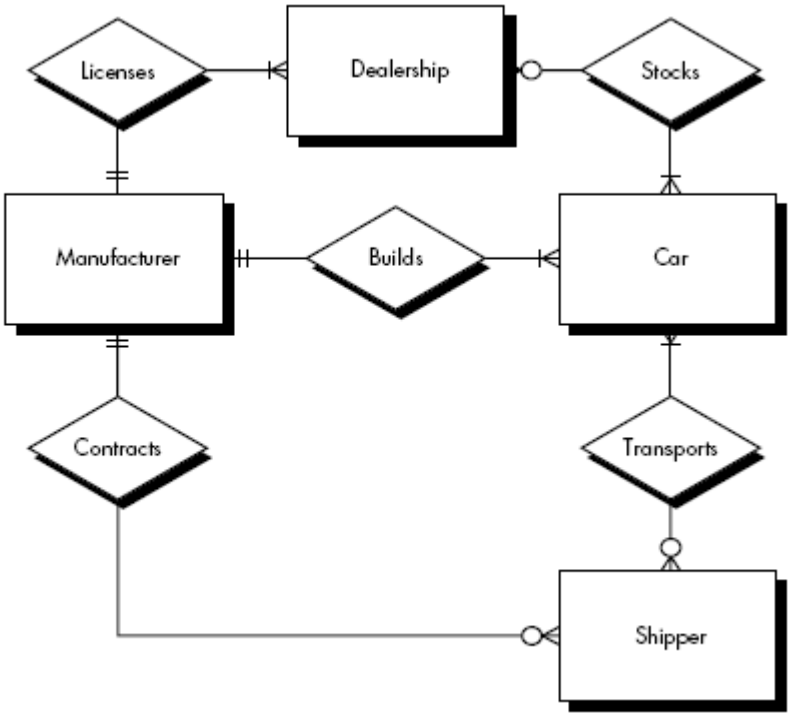


FIGURE 12.7
An expanded ERD



Relationships

Data objects are connected to one another in different ways. Consider the two data objects, **person** and **car.** These objects can be represented using the simple notation

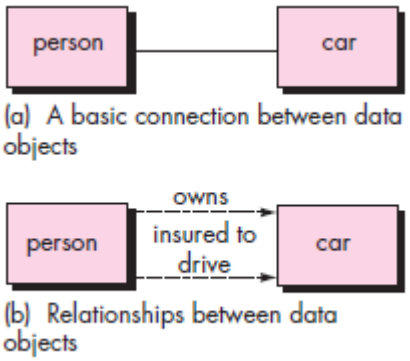


Fig 6.8. Relationships between data objects

illustrated in Figure 6.8a. A connection is established between **person** and **car** because the two objects are related. But what are the relationships? To determine the answer, you should understand the role of people (owners, in this case) and cars within the context of the software to be built. You can establish a set of object/ relationship pairs that define the relevant relationships. For example,

- A person *owns* a car.
- A person *is insured to drive* a car.

The relationships *owns* and *insured to drive* define the relevant connections between **person** and **car.** Figure 6.8b illustrates these object-relationship pairs graphically. The arrows noted in Figure 6.8b provide important information about the directionality of the relationship and often reduce ambiguity or misinterpretations.

FLOW ORIENTED MODELING

Although data flow-oriented modeling is perceived as an outdated technique by some software engineers, it continues to be one of the most widely used requirements analysis notations in use today.1 Although the *data flow diagram* (DFD) and related diagrams and information are not a

formal part of UML, they can be used to complement UML diagrams and provide additional insight into system requirements and flow.

The DFD takes an input-process-output view of a system. That is, data objects flow into the software, are transformed by processing elements, and resultant data objects flow out of the software. Data objects are represented by labeled arrows, and transformations are represented by circles (also called bubbles). The DFD is presented in a hierarchical fashion. That is, the first data flow model (sometimes called a level 0 DFD or *context diagram*) represents the system as a whole. Subsequent data flow diagrams refine the context diagram, providing increasing detail with each subsequent level.

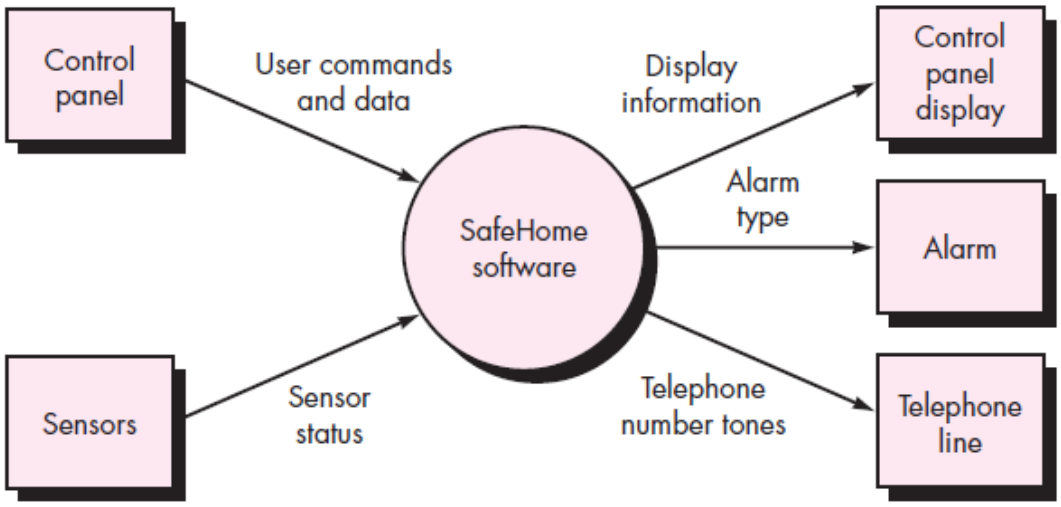


Fig 7.1.Context-level DFD for the *SafeHome* security function.

CREATING A DATA FLOW MODEL

The data flow diagram enables you to develop models of the information domain and functional domain. As the DFD is refined into greater levels of detail, you perform an implicit functional decomposition of the system. At the same time, the DFD refinement results in a corresponding refinement of data as it moves through the processes that embody the application.

A few simple guidelines can aid immeasurably during the derivation of a data flow diagram: (1) the level 0 data flow diagram should depict the software/system as a single bubble; (2) primary input and output should be carefully noted; (3) refinement should begin by isolating candidate

processes, data objects, and data stores to be represented at the next level; (4) all arrows and bubbles should be labeled with meaningful names; (5) *information flow continuity* must be maintained from level to level,2 and (6) one bubble at a time should be refined. There is a natural tendency to overcomplicate the data flow diagram. This occurs when you attempt to show too much detail too early or represent procedural aspects of the software in lieu of information flow.

To illustrate the use of the DFD and related notation, we again consider the *SafeHome* security function. A level 0 DFD for the security function is shown in Figure 7.1. The primary *external entities* (boxes) produce information for use by the system and consume information generated by the system. The labeled arrows represent data objects or data object hierarchies. For example, **user commands and data** encompasses all configuration commands, all activation/deactivation commands, all miscellaneous interactions, and all data that are entered to qualify or expand a command.

The level 0 DFD must now be expanded into a level 1 data flow model. But how do we proceed? Following an approach suggested in Chapter 6, you should apply a "grammatical parse" to the use case narrative that describes the context-level bubble. That is, we isolate all nouns (and noun phrases) and verbs (and verb phrases) in a *SafeHome* processing narrative derived during the first requirements gathering meeting. The SafeHome security function *enables* the homeowner to *configure* the security system when it is *installed, monitors* all sensors *connected* to the security system, and *interacts* with the homeowner through the Internet, a PC, or a control panel.

During installation, the *SafeHome* PC is used to *program* and *configure* the system. Each sensor is assigned a number and type, a master password is programmed for *arming* and *disarming* the system, and telephone number(s) are *input* for *dialing* when a sensor event occurs.

When a sensor event is *recognized*, the software *invokes* an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, *provides* information about the location, *reporting* the nature of the event that has been detected. The telephone number will be *redialed* every 20 seconds until telephone connection is *obtained.* The homeowner *receives* security information via a control panel, the PC, or a browser, collectively called an interface. The interface *displays* prompting messages and system status information on the control panel, the PC, or the browser window. Homeowner interaction takes the following form . . .

Referring to the grammatical parse, verbs are *SafeHome* processes and can be represented as bubbles in a subsequent DFD. Nouns are either external entities (boxes), data or control objects

(arrows), or data stores (double lines). Nouns and verbs can be associated with one another (e.g., each sensor is assigned a number and type; therefore number and type are attributes of the data object **sensor**).

Therefore, by performing a grammatical parse on the processing narrative for a bubble at any DFD level, you can generate much useful information about how to proceed with the refinement to the next level. Using this information, a level 1 DFD is shown in Figure 7.2. The context level process shown in Figure 7.1 has been expanded into six processes derived from an examination of the grammatical parse. Similarly, the information flow between processes at level 1 has been derived from the parse. In addition, information flow continuity is maintained between levels 0 and 1.

The processes represented at DFD level 1 can be further refined into lower levels. For example, the process *monitor sensors* can be refined into a level 2 DFD as shown in Figure 7.3. Note once again that information flow continuity has been maintained between levels.

The refinement of DFDs continues until each bubble performs a simple function. That is, until the process represented by the bubble performs a function that would be easily implemented as a program component. In Chapter 8, I discuss a concept, called *cohesion*, that can be used to assess the processing focus of a given function. For now, we strive to refine DFDs until each bubble is "single-minded."

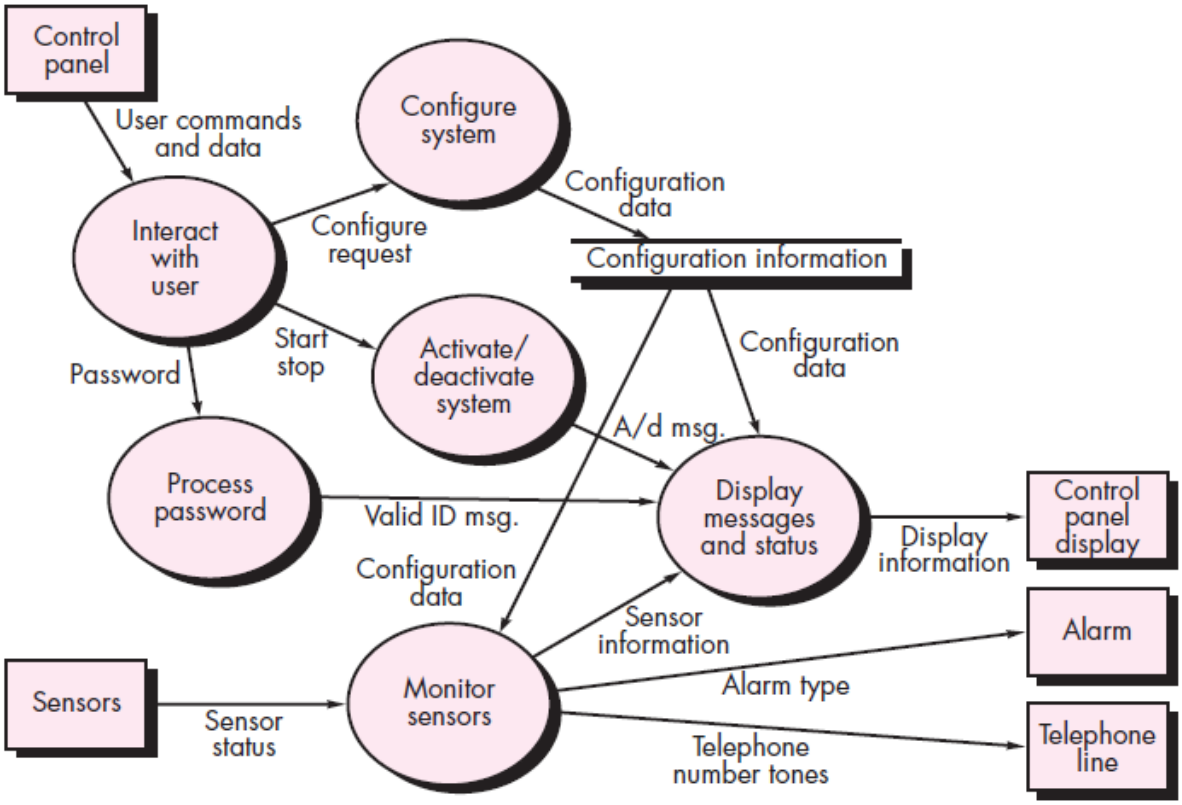


Fig 2.2. Level 1 DFD for *SafeHome* security function.

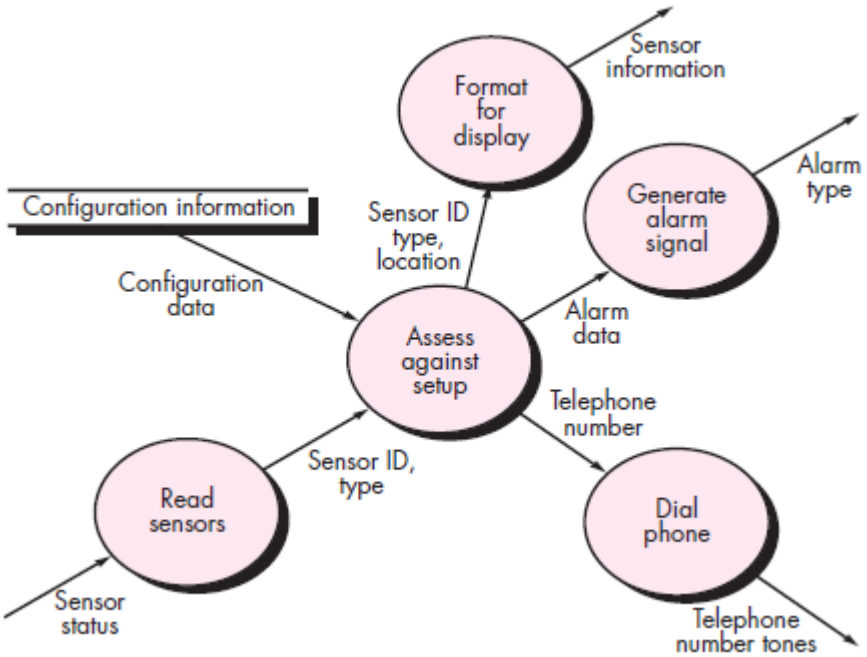


Fig 2.3. Level 2 DFD that refines the *monitor sensors* process

Creating a Control Flow Model

For some types of applications, the data model and the data flow diagram are all that is necessary to obtain meaningful insight into software requirements. As I have already noted, however, a large class of applications are "driven" by events rather than data, produce control information rather than reports or displays, and process information with heavy concern for time and performance. Such applications require the use of *control flow modeling* in addition to data flow modeling.

I have already noted that an event or control item is implemented as a Boolean value (e.g., true or false, on or off, 1 or 0) or a discrete list of conditions (e.g., empty, jammed, full). To select potential candidate events, the following guidelines are suggested:

• List all sensors that are "read" by the software.

• List all interrupt conditions.

• List all "switches" that are actuated by an operator.

• List all data conditions.

• Recalling the noun/verb parse that was applied to the processing narrative, review all "control items" as possible control specification inputs/outputs.

• Describe the behavior of a system by identifying its states, identify how each state is reached, and define the transitions between states. • Focus on possible omissions—a very common error in specifying control; for example, ask: "Is there any other way I can get to this state or exit from it?"

Among the many events and control items that are part of *SafeHome* software are **sensor event** (i.e., a sensor has been tripped), **blink flag** (a signal to blink the display), and **start/stop switch** (a signal to turn the system on or off).

THE CONTROL SPECIFICATION

A *control specification* (CSPEC) represents the behavior of the system (at the level from which it has been referenced) in two different ways.3 The CSPEC contains a state diagram that is a sequential specification of behavior. It can also contain a program activation table—a combinatorial specification of behavior.

Figure 2.4 depicts a preliminary state diagram4 for the level 1 control flow model for *SafeHome.* The diagram indicates how the system responds to events as it traverses the four states defined at this level. By reviewing the state diagram, you can determine the behavior of the system and, more important, ascertain whether there are "holes" in the specified behavior.

For example, the state diagram (Figure 7.4) indicates that the transitions from the **Idle** state can occur if the system is reset, activated, or powered off. If the system is

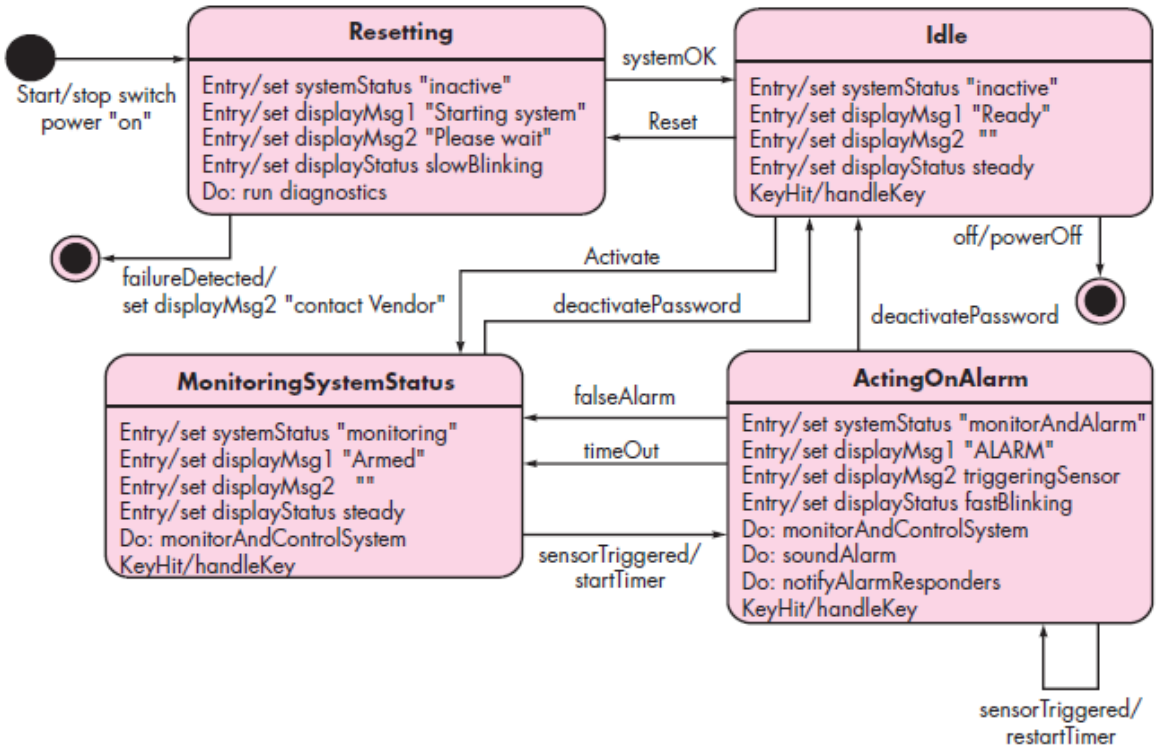


Fig 2.4. State diagram for *SafeHome* security function

activated (i.e., alarm system is turned on), a transition to the **Monitoring- SystemStatus** state occurs, display messages are changed as shown, and the process *monitorAndControlSystem* is invoked. Two transitions occur out of the **MonitoringSystemStatus** state—(1) when the system is deactivated, a transition occurs back to the **Idle** state; (2) when a sensor is triggered into the **ActingOnAlarm** state. All transitions and the content of all states are considered during the review.

A somewhat different mode of behavioral representation is the process activation table. The PAT represents information contained in the state diagram in the context of processes, not states. That is, the table indicates which processes (bubbles) in the flow model will be invoked when an event occurs. The PAT can be used as a guide for a designer who must build an executive that controls the processes represented at this level. A PAT for the level 1 flow model of *SafeHome* software is shown in Figure 2.5.

The CSPEC describes the behavior of the system, but it gives us no information about the inner working of the processes that are activated as a result of this behavior.

THE PROCESS SPECIFICATION

The *process specification* (PSPEC) is used to describe all flow model processes that appear at the final level of refinement. The content of the process specification can

input events						
sensor event	0	0	0	0	1	0
blink flag	0	0	1	1	0	0
start stop switch	0	1	0	0	0	0
display action status complete	0	0	0	1	0	0
in-progress	0	0	1	0	0	0
time out	0	0	0	0	0	1
output						
alarm signal	0	0	0	0	1	0
process activation						
monitor and control system	0	1	0	0	1	1
activate/deactivate system	0	1	0	0	0	0
display messages and status	1	0	1	1	1	1
interact with user	1	0	0	1	0	1

Fig 2.5. Process activation table for *SafeHome* security function

include narrative text, a program design language (PDL) description5 of the process algorithm, mathematical equations, tables, or UML activity diagrams. By providing a PSPEC to accompany each bubble in the flow model, you can create a "mini-spec" that serves as a guide for design of the software component that will implement the bubble. To illustrate the use of the PSPEC, consider the *process password* transform represented in the flow model for *SafeHome* (Figure 7.2). The PSPEC for this function might take the form:

PSPEC: process password (at control panel). The *process password* transform performs password validation at the control panel for the *SafeHome* security function. *Process password* receives a four-digit password from the *interact with user* function. The password is first compared to the master password stored within the system. If the master password matches, <valid id message = true> is passed to the *message and status display* function. If the master password does not match, the four digits are compared to a table of secondary passwords (these may be assigned to house guests and/or workers who require entry to the home when the owner is not present). If the password matches an entry within the table, <valid id message = true> is

passed to the *message and status display function*. If there is no match, <valid id message = false> is passed to the message and status display function. If additional algorithmic detail is desired at this stage, a program design language representation may also be included as part of the PSPEC. However, many believe that the PDL version should be postponed until component design commences.

CREATING A BEHAVIORAL MODEL.

The modeling notation represents static elements of the requirements model. It is now time to make a transition to the dynamic behavior of the system or product. To accomplish this, you can represent the behavior of the system as a function of specific events and time.

The *behavioral model* indicates how software will respond to external events or stimuli. To create the model, you should perform the following steps:

1. Evaluate all use cases to fully understand the sequence of interaction within the system.

2. Identify events that drive the interaction sequence and understand how these events relate to specific objects.

3. Create a sequence for each use case.

4. Build a state diagram for the system.

5. Review the behavioral model to verify accuracy and consistency.

Each of these steps is discussed in the sections that follow.

Identifying Events with the Use Case

In Chapter 6 you learned that the use case represents a sequence of activities that involves actors and the system. In general, an event occurs whenever the system and an actor exchange information. In Section 7.2.3, I indicated that an event is *not* the information that has been exchanged, but rather the fact that information has been exchanged.

A use case is examined for points of information exchange. To illustrate, we reconsider the use case for a portion of the *SafeHome* security function.

The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.

The underlined portions of the use case scenario indicate events. An actor should be identified for each event; the information that is exchanged should be noted, and any conditions or constraints should be listed.

As an example of a typical event, consider the underlined use case phrase "homeowner uses the keypad to key in a four-digit password." In the context of the requirements model, the object, **Homeowner,**7 transmits an event to the object **ControlPanel.** The event might be called *password entered*.

The information transferred is the four digits that constitute the password, but this is not an essential part of the behavioral model. It is important to note that some events have an explicit impact on the flow of control of the use case, while others have no direct impact on the flow of control. For example, the event *password entered* does not explicitly change the flow of control of the use case, but the results of the event *password compared* (derived from the interaction "password is compared with the valid password stored in the system") will have an explicit impact on the information and control flow of the *SafeHome* software.

Once all events have been identified, they are allocated to the objects involved. Objects can be responsible for generating events (e.g., **Homeowner** generates the *password entered* event) or recognizing events that have occurred elsewhere (e.g., **ControlPanel** recognizes the binary result of the *password compared* event).

State Representations

In the context of behavioral modeling, two different characterizations of states must be considered: (1) the state of each class as the system performs its function and (2) the state of the system as observed from the outside as the system performs its function.8

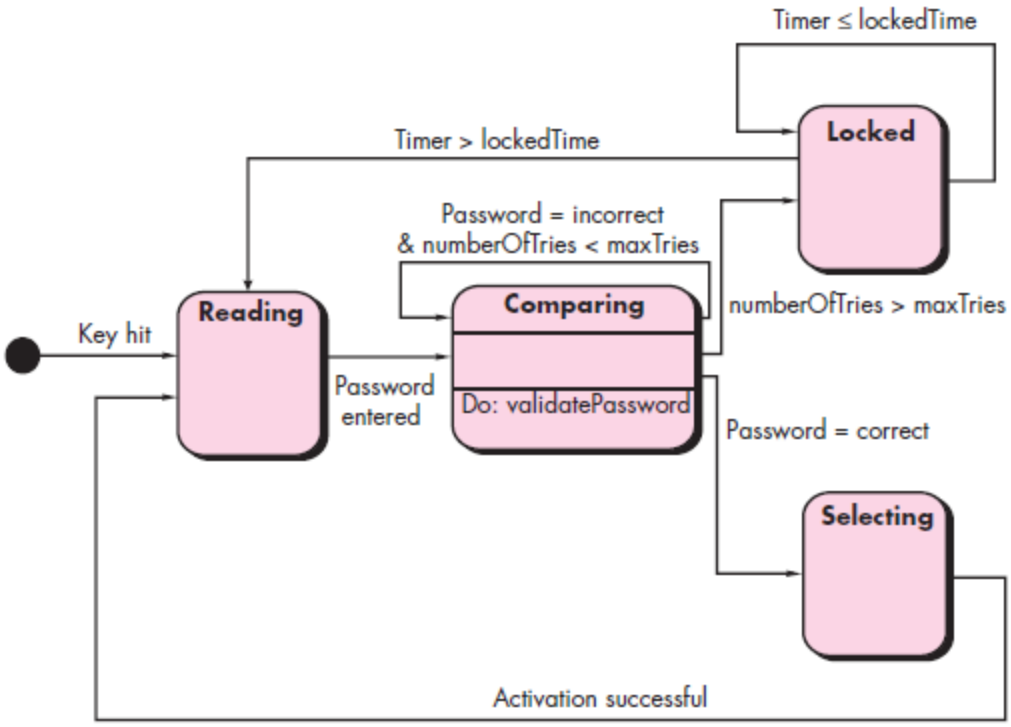
The state of a class takes on both passive and active characteristics [Cha93]. A *passive state* is simply the current status of all of an object's attributes. For example, the passive state of the class **Player** (in the video game application discussed in Chapter 6) would include the current position and orientation attributes of **Player** as well as other features of **Player** that are relevant to the game (e.g., an attribute that indicates magic wishes remaining).

The *active state* of an object indicates the current status of the object as it undergoes a continuing transformation or processing. The class **Player** might have the following active states: *moving, at rest, injured, being cured; trapped, lost,* and so forth. An event (sometimes called a *trigger*) must occur to force an object to make a transition from one active state to another.

Two different behavioral representations are discussed in the paragraphs that follow. The first indicates how an individual class changes state based on external events and the second shows the behavior of the software as a function of time. **State diagrams for analysis classes.**

One component of a behavioral model is a UML state diagram9 that represents active states for each class and the events (triggers) that cause changes between these active states. Figure 7.6 illustrates a state diagram for the **ControlPanel** object in the *SafeHome* security function.

Each arrow shown in Figure 7.6 represents a transition from one active state of an object to another. The labels shown for each arrow represent the event that



State diagram for the ControlPanel Class

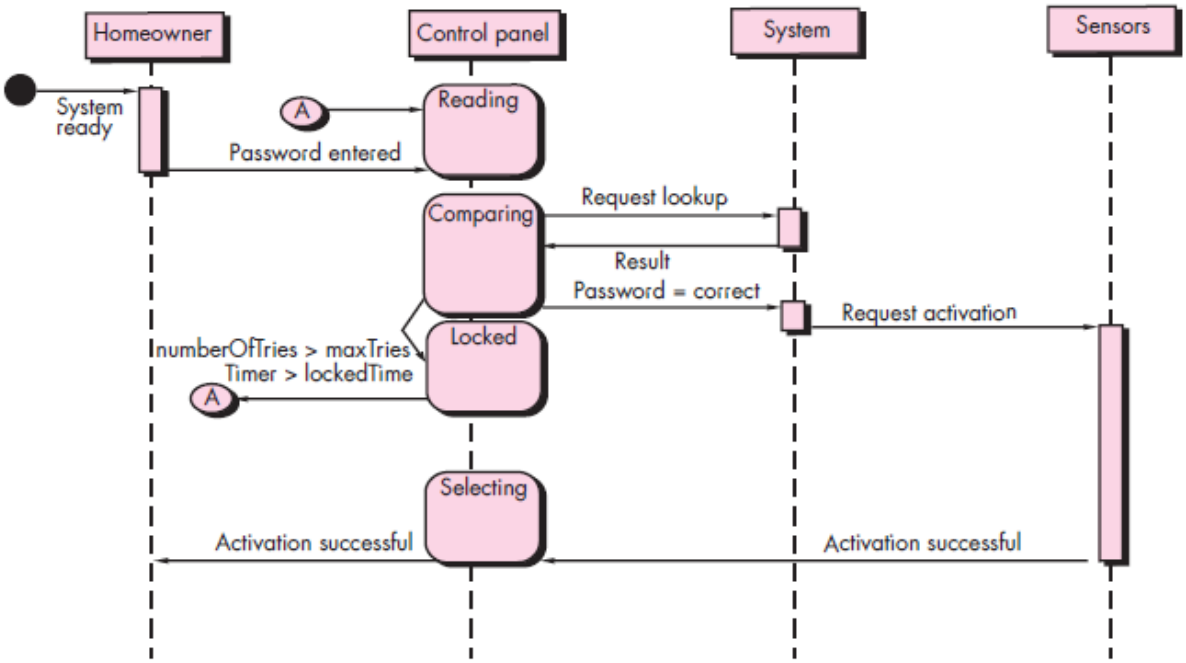
triggers the transition. Although the active state model provides useful insight into the "life history" of an object, it is possible to specify additional information to provide more depth in understanding the behavior of an object. In addition to specifying the event that causes the transition to occur, you can specify a guard and an action [Cha93]. A *guard* is a Boolean condition that must be satisfied in order for the transition to occur. For example, the guard for the transition from the "reading" state to the "comparing" state in Figure 7.6 can be determined by examining the use case: if (password input _ 4 digits) then compare to stored password

In general, the guard for a transition usually depends upon the value of one or more attributes of an object. In other words, the guard depends on the passive state of the object.

An *action* occurs concurrently with the state transition or as a consequence of it and generally involves one or more operations (responsibilities) of the object. For example, the action connected to the *password entered* event (Figure 7.6) is an operation named *validatePassword()* that accesses a **password** object and performs a digit-by-digit comparison to validate the entered password.

Sequence diagrams.

The second type of behavioral representation, called a *sequence diagram* in UML, indicates how events cause transitions from object to object. Once events have been identified by examining a use case, the modeler



Sequence diagram (partial) for the *SafeHome* security function creates a sequence diagram—a representation of how events cause flow from one object to another as a function of time. In essence, the sequence diagram is a shorthand version of the use case. It represents key classes and the events that cause behavior to flow from class to class.

Figure 7.7 illustrates a partial sequence diagram for the *SafeHome* security function. Each of the arrows represents an event (derived from a use case) and indicates how the event channels behavior between *SafeHome* objects. Time is measured vertically (downward), and the narrow vertical rectangles represent time spent in processing an activity. States may be shown along a vertical time line.

The first event, *system ready*, is derived from the external environment and channels behavior to the **Homeowner** object. The homeowner enters a password. A *request lookup* event is passed to **System,** which looks up the password in a simple database and returns a *result* (*found* or *not found*) to **ControlPanel** (now in the *comparing* state). A valid password results in a *password=correct* event to **System,** which activates **Sensors** with a *request activation* event. Ultimately, control is passed back to the homeowner with the *activation successful* event.

Once a complete sequence diagram has been developed, all of the events that cause transitions between system objects can be collated into a set of input events and output events (from an object). This information is useful in the creation of an effective design for the system to be built.

POSSIBLE QUESTIONS

PART – B

1. Explain Structure Analysis Model?
2. Describe about the Requirement Analysis.
3. Explain in detail about data modeling concepts with examples.
4. Describe flow-oriented modeling with examples.
5. Describe about Process Specification and Control Specification.
6. Develop state diagram and sequence diagram that could serve as a basis for understanding the requirements for a SafeHome Security function.
7. Elucidate the steps to create a behavioral model.

UNIT-II

SYLLABUS

Building the Analysis Model: Requirements Analysis-Analysis Modeling Approaches-Data Modeling Concepts: Data Objects-Date attributes-Relationships Cardinality and Modality-Flow Oriented Modeling: Creating Data Flow Model-Creating a Control Flow Model-The Control Specification-The Process Specification- Creating a Behavioral Model.

BUILDING THE ANALYSIS MODEL:

At a technical level, software engineering begins with a series of modeling tasks that lead to a specification of requirements and a design representation for the software to be built. The requirements model— actually a set of models—is the first technical representation of a system.

REQUIREMENTS ANALYSIS

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows you (regardless of whether you're called a *software engineer*, an *analyst*, or a *modeler*) to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering.

The requirements modeling action results in one or more of the following types of models:

• *Scenario-based models* of requirements from the point of view of various system "actors"

• *Data models* that depict the information domain for the problem

• *Class-oriented models* that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements

• *Flow-oriented models* that represent the functional elements of the system and how they transform data as it moves through the system

• *Behavioral models* that depict how the software behaves as a consequence of external "events"

These models provide a software designer with information that can be translated to architectural, interface, and component-level designs. Finally, the requirements model (and the

software requirements specification) provides the developer and the customer with the means to assess quality once software is built.

Scenario-based modeling—a technique that is growing increasingly popular throughout the software engineering community; *data modeling*—a more specialized technique that is particularly appropriate when an application must create or manipulate a complex information space; and *class modeling*—a representation of the object-oriented classes and the resultant collaborations that allow a system to function.

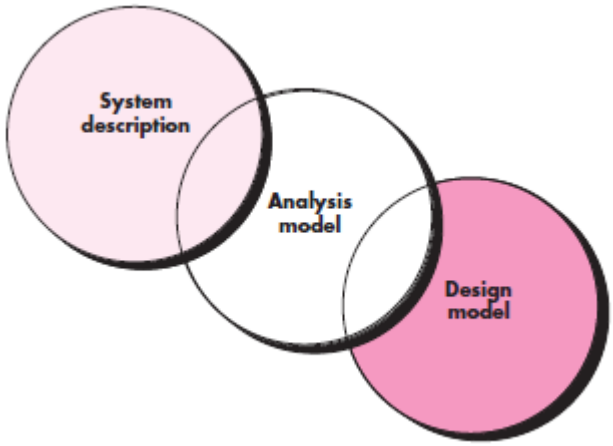


Fig 2.1. The requirements model as a bridge between the system description and the design model

Overall Objectives and Philosophy

Throughout requirements modeling, your primary focus is on *what*, not *how*. What user interaction occurs in a particular circumstance, what objects does the system manipulate, what functions must the system perform, what behaviors does the system exhibit, what interfaces are defined, and what constraints apply?

The customer may be unsure of precisely what is required for certain aspects of the system. The developer may be unsure that a specific approach will properly accomplish function and performance. These realities mitigate in favor of an iterative approach to requirements analysis and modeling. The analyst should model what is known and use that model as the basis for design of the software increment.

The requirements model must achieve three primary objectives:

(1) to describe what the customer requires,

(2) to establish a basis for the creation of a software design, and

(3) to define a set of requirements that can be validated once the software is built.

The analysis model bridges the gap between a system-level description that describes overall system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design that describes the software's application architecture, user interface, and component-level structure. This relationship is illustrated in Figure 2.1.

It is important to note that all elements of the requirements model will be directly traceable to parts of the design model. A clear division of analysis and design tasks between these two important modeling activities is not always possible. Some design invariably occurs as part of analysis, and some analysis will be conducted during design.

Analysis Rules of Thumb

Arlow and Neustadt suggest a number of worthwhile rules of thumb that should be followed when creating the analysis model:

- *The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.* "Don't get bogged down in details" that try to explain how the system will work.
- *Each element of the requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system.*
- *Delay consideration of infrastructure and other nonfunctional models until design.* That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.
- *Minimize coupling throughout the system.* It is important to represent relationships between classes and functions. However, if the level of "interconnectedness" is extremely high, effort should be made to reduce it.

- *Be certain that the requirements model provides value to all stakeholders.* Each constituency has its own use for the model. For example, business stakeholders should use the model to validate requirements; designers should use the model as a basis for design; QA people should use the model to help plan acceptance tests.

- *Keep the model as simple as it can be.* Don't create additional diagrams when they add no new information. Don't use complex notational forms, when a simple list will do.

Domain Analysis

In the discussion of requirements engineering, I noted that analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows you to recognize and apply them to solve common problems, the creation of the analysis model is expedited. More important, the likelihood of applying design patterns and executable software components grows dramatically. This improves time-to-market and reduces development costs.

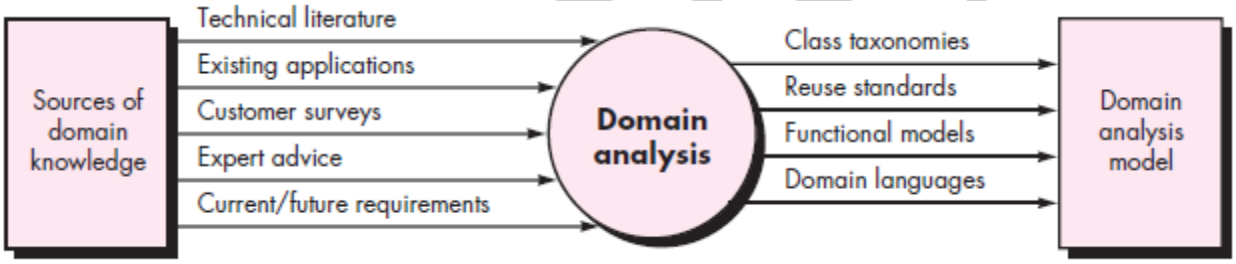


Fig 2.2. Input and output for domain analysis

But how are analysis patterns and classes recognized in the first place? Who defines them, categorizes them, and readies them for use on subsequent projects? The answers to these questions lie in *domain analysis.* Firesmith describes domain analysis in the following way:

Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain. . . . [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks.

The "specific application domain" can range from avionics to banking, from multimedia video games to software embedded within medical devices. The goal of domain analysis is straightforward: to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.

Using terminology that was introduced earlier in this book, domain analysis may be viewed as an umbrella activity for the software process. By this I mean that domain analysis is an ongoing software engineering activity that is not connected to any one software project. In a way, the role of a domain analyst is similar to the role of a master toolsmith in a heavy manufacturing environment. The job of the toolsmith is to design and build tools that may be used by many people doing similar but not necessarily the same jobs.

The role of the domain analyst5 is to discover and define analysis patterns, analysis classes, and related information that may be used by many people working on similar but not necessarily the same applications. Figure 6.2 [Ara89] illustrates key inputs and outputs for the domain analysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain.

ANALYSIS MODELING APPROACHES

Requirements Modeling Approaches

One view of requirements modeling, called *structured analysis*, considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.

A second approach to analysis modeling, called *object-oriented analysis*, focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process are predominantly object oriented.

Although the requirements model proposed in this book combines features of both approaches, software teams often choose one approach and exclude all representations from the other.

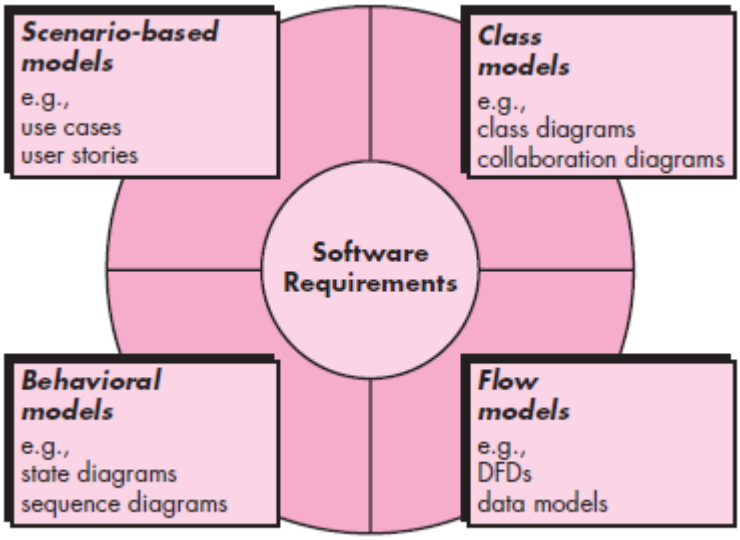


Fig 2.3. Elements of the analysis model

The question is not which is best, but rather, what combination of representations will provide stakeholders with the best model of software requirements and the most effective bridge to software design. Each element of the requirements model (Figure 2.3) presents the problem from a different point of view. Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used.

Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. Behavioral elements depict how external events change the state of the system or the classes that reside within it. Finally, flow-oriented elements represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions.

Analysis modeling leads to the derivation of each of these modeling elements. However, the specific content of each element (i.e., the diagrams that are used to construct the element and the model) may differ from project to project. As we have noted a number of times in this book, the software team must work to keep it simple. Only those modeling elements that add value to the model should be used.

DATA MODELING CONCEPTS:

If software requirements include the need to create, extend, or interface with a database or if complex data structures must be constructed and manipulated, the software team may choose to create a *data model* as part of overall requirements modeling. A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships. The *entity-relationship diagram* (ERD) addresses these issues and represents all data objects that are entered, stored, transformed, and produced within an application.

Data Objects

A *data object* is a representation of composite information that must be understood by software. By *composite information*, I mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but **dimensions** (incorporating height, width, and depth) could be defined as an object.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a **person** or a **car** can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The description of the data object incorporates the data object and all of its attributes.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data. Therefore, the data object can be represented as a table as shown in Figure 6.7. The headings in the table reflect attributes of the object. In this case, a car is defined in terms of make, model, ID number, body type, color, and owner. The body of the table represents specific instances of the data object. For example, a Chevy Corvette is an instance of the data object car.

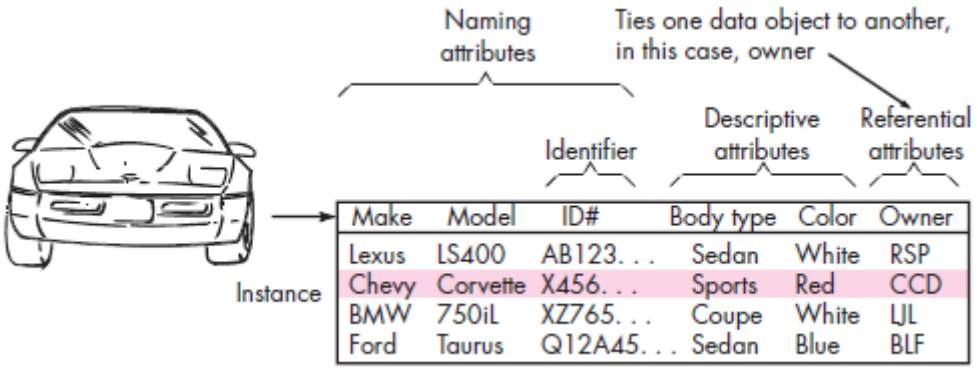
DATA ATTRIBUTES

Data attributes define the properties of a data object and take on one of three different characteristics. They can be used to

(1) name an instance of the data object,

(2) describe the instance, or

(3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier—that is, the identifier



Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123. . .	Sedan	White	RSP
Chevy	Corvette	X456. . .	Sports	Red	CCD
BMW	750iL	XZ765. . .	Coupe	White	LJL
Ford	Taurus	Q12A45. . .	Sedan	Blue	BLF

Fig 6.7 Tabular representation of data objects

attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object **car,** a reasonable identifier might be the ID number.

The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context. The attributes for **car** might serve well for an application that would be used by a department of motor vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software. In the latter case, the attributes for **car** might also include ID number, body type, and color, but many additional attributes (e.g., interior code, drive train type, trim package designator, transmission type) would have to be added to make **car** a meaningful object in the manufacturing control context.

Data Objects, Attributes, and Relationships

The data model consists of three interrelated pieces of information: the data object, the attributes that describe the data object, and the relationships that connect data objects to one another.

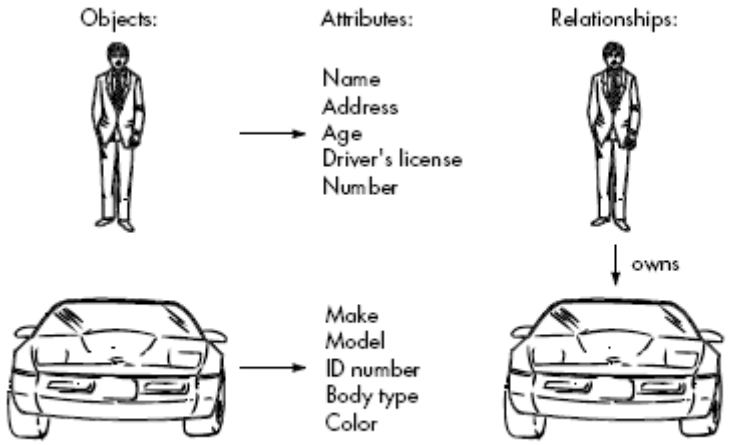


FIGURE 12.2 Data objects, attributes and relationships

Relationships. Data objects are connected to one another in different ways. Consider two data objects, **book** and **bookstore**. These objects can be represented using the simple notation illustrated in Figure 12.4a. A connection is established between **book** and **bookstore** because the two objects are related. But what are the relationships? To determine the answer, we must understand the role of books and bookstores within the context of the software to be built. We can define a set of object/relationship pairs that define the relevant relationships. For example,

A bookstore orders books.

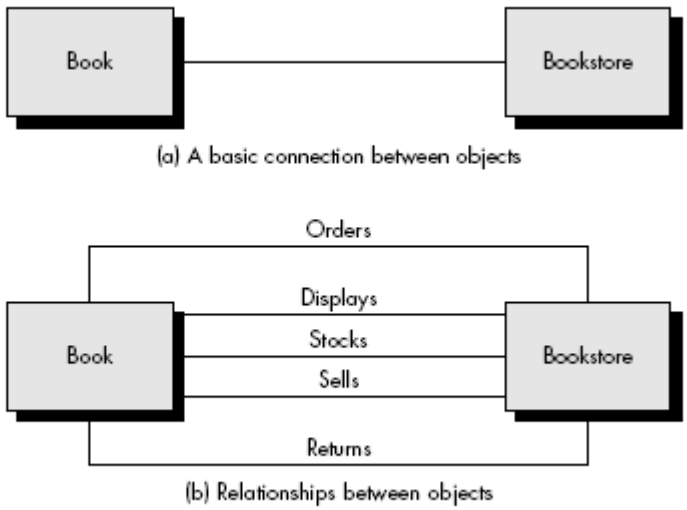
• A bookstore displays books.

• A bookstore stocks books.

• A bookstore sells books.

• A bookstore returns books.

FIGURE 12.4
Relationships



The relationships orders, displays, stocks, sells, and returns define the relevant connections between book and bookstore. Figure 12.4b illustrates these object/relationship pairs graphically. It is important to note that object/relationship pairs are bidirectional. That is, they can be read in either direction. A bookstore orders books or books are ordered by a bookstore.

Cardinality and Modality

• Cardinality is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object].

• Cardinality is usually expressed as simply 'one' or 'many.'

• Cardinality defines "the maximum number of objects that can participate in a relationship".

• It does not, however, provide an indication of whether or not a particular data object must participate in the relationship. To specify this information, the data model adds modality to the object/relationship pair

Modality

• The modality of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional.

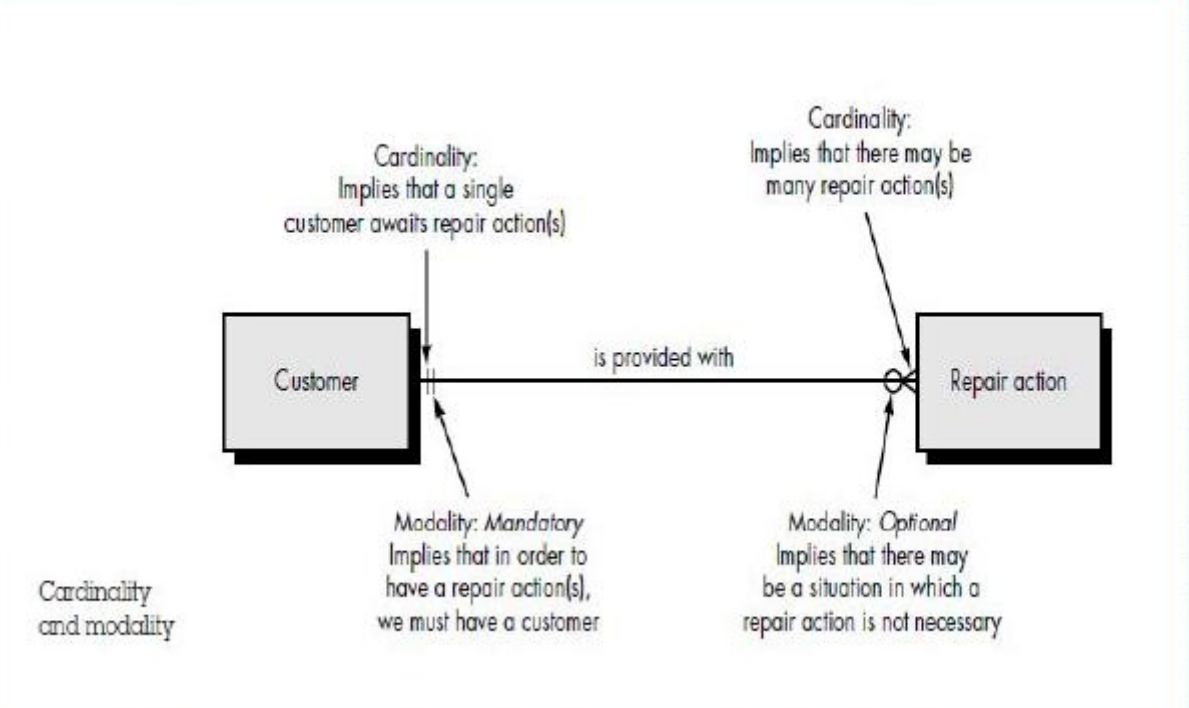
• The modality is 1 if an occurrence of the relationship is mandatory.

Example

• Consider software that is used by a local telephone company to process requests for field service. A customer indicates that there is a problem. If the problem is diagnosed as relatively

simple, a single repair action occurs. However, if the problem is complex, multiple repair actions may be required.

• Following figure illustrates the relationship, cardinality, and modality between the data objects customer and repair action.



Cardinality and modality

Entity/Relationship Diagrams

The object/relationship pair is the cornerstone of the data model. These pairs can be represented graphically using the *entity/relationship diagram*. The ERD was originally proposed by Peter Chen for the design of relational database systems and has been extended by others. A set of primary components are identified for the ERD: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships.

Data objects are represented by a labeled rectangle. Relationships are indicated with a labeled line connecting objects. In some variations of the ERD, the connecting line contains a diamond that is labeled with the relationship. Connections between data objects and relationships are established using a variety of special symbols that indicate cardinality and modality. The relationship between the data objects **car** and **manufacturer** would be represented as shown in Figure 12.6. One manufacturer builds one or many cars. Given the context implied by the ERD, the specification of the data object **car** (data object table in Figure 12.6) would be radically different from the earlier specification (Figure 12.3). By examining the symbols at the end of the

connection line between objects, it can be seen that the modality of both occurrences is mandatory (the vertical lines).

Expanding the model, we represent a grossly oversimplified ERD (Figure 12.7) of the distribution element of the automobile business. New data objects, **shipper** and **dealership,** are introduced. In addition, new relationships—*transports, contracts, licenses,* and *stocks*—indicate how the data objects shown in the figure associate with one another

FIGURE 12.6
A simple ERD and data object table (Note: In this ERD the relationship *builds* is indicated by a diamond)

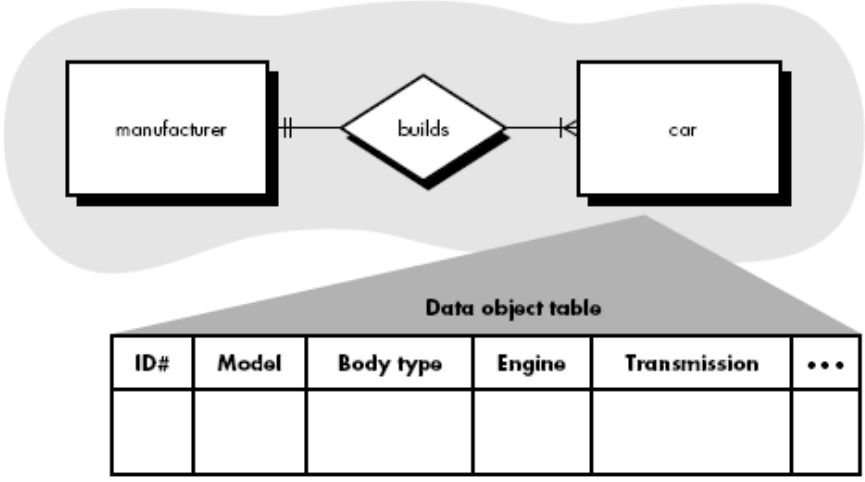
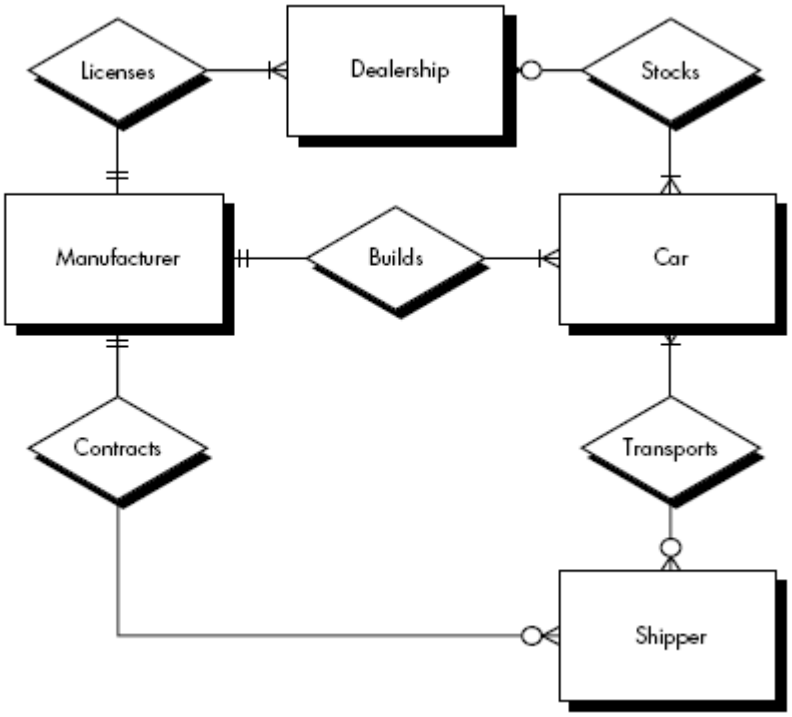


FIGURE 12.7
An expanded ERD



Relationships

Data objects are connected to one another in different ways. Consider the two data objects, **person** and **car.** These objects can be represented using the simple notation

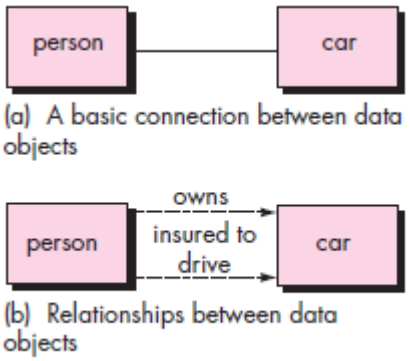


Fig 6.8. Relationships between data objects

illustrated in Figure 6.8a. A connection is established between **person** and **car** because the two objects are related. But what are the relationships? To determine the answer, you should understand the role of people (owners, in this case) and cars within the context of the software to be built. You can establish a set of object/ relationship pairs that define the relevant relationships. For example,

- A person *owns* a car.
- A person *is insured to drive* a car.

The relationships *owns* and *insured to drive* define the relevant connections between **person** and **car.** Figure 6.8b illustrates these object-relationship pairs graphically. The arrows noted in Figure 6.8b provide important information about the directionality of the relationship and often reduce ambiguity or misinterpretations.

FLOW ORIENTED MODELING

Although data flow-oriented modeling is perceived as an outdated technique by some software engineers, it continues to be one of the most widely used requirements analysis notations in use today.1 Although the *data flow diagram* (DFD) and related diagrams and information are not a

formal part of UML, they can be used to complement UML diagrams and provide additional insight into system requirements and flow.

The DFD takes an input-process-output view of a system. That is, data objects flow into the software, are transformed by processing elements, and resultant data objects flow out of the software. Data objects are represented by labeled arrows, and transformations are represented by circles (also called bubbles). The DFD is presented in a hierarchical fashion. That is, the first data flow model (sometimes called a level 0 DFD or *context diagram*) represents the system as a whole. Subsequent data flow diagrams refine the context diagram, providing increasing detail with each subsequent level.

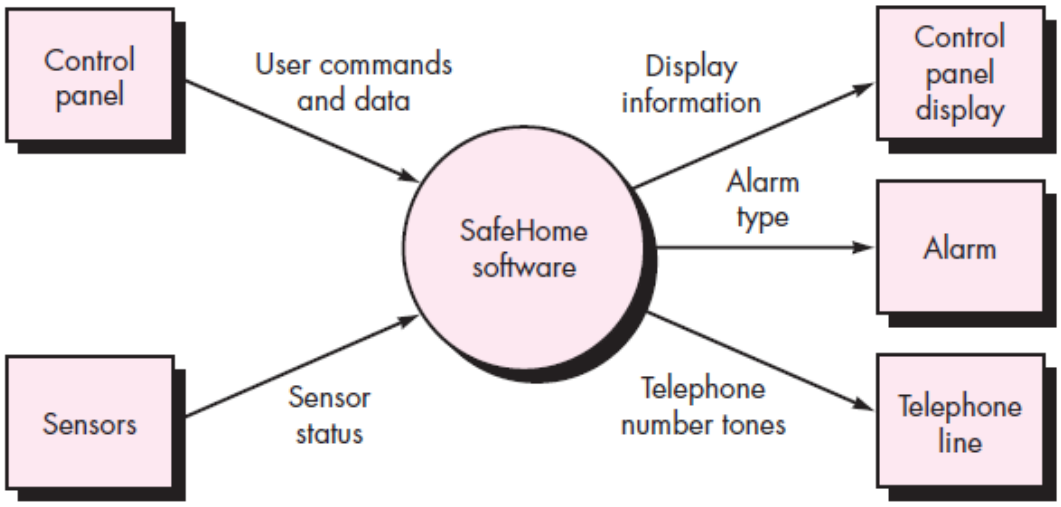


Fig 7.1.Context-level DFD for the *SafeHome* security function.

CREATING A DATA FLOW MODEL

The data flow diagram enables you to develop models of the information domain and functional domain. As the DFD is refined into greater levels of detail, you perform an implicit functional decomposition of the system. At the same time, the DFD refinement results in a corresponding refinement of data as it moves through the processes that embody the application.

A few simple guidelines can aid immeasurably during the derivation of a data flow diagram: (1) the level 0 data flow diagram should depict the software/system as a single bubble; (2) primary input and output should be carefully noted; (3) refinement should begin by isolating candidate

processes, data objects, and data stores to be represented at the next level; (4) all arrows and bubbles should be labeled with meaningful names; (5) *information flow continuity* must be maintained from level to level,2 and (6) one bubble at a time should be refined. There is a natural tendency to overcomplicate the data flow diagram. This occurs when you attempt to show too much detail too early or represent procedural aspects of the software in lieu of information flow.

To illustrate the use of the DFD and related notation, we again consider the *SafeHome* security function. A level 0 DFD for the security function is shown in Figure 7.1. The primary *external entities* (boxes) produce information for use by the system and consume information generated by the system. The labeled arrows represent data objects or data object hierarchies. For example, **user commands and data** encompasses all configuration commands, all activation/deactivation commands, all miscellaneous interactions, and all data that are entered to qualify or expand a command.

The level 0 DFD must now be expanded into a level 1 data flow model. But how do we proceed? Following an approach suggested in Chapter 6, you should apply a "grammatical parse" to the use case narrative that describes the context-level bubble. That is, we isolate all nouns (and noun phrases) and verbs (and verb phrases) in a *SafeHome* processing narrative derived during the first requirements gathering meeting. The SafeHome security function *enables* the homeowner to *configure* the security system when it is *installed, monitors* all sensors *connected* to the security system, and *interacts* with the homeowner through the Internet, a PC, or a control panel.

During installation, the *SafeHome* PC is used to *program* and *configure* the system. Each sensor is assigned a number and type, a master password is programmed for *arming* and *disarming* the system, and telephone number(s) are *input* for *dialing* when a sensor event occurs.

When a sensor event is *recognized*, the software *invokes* an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, *provides* information about the location, *reporting* the nature of the event that has been detected. The telephone number will be *redialed* every 20 seconds until telephone connection is *obtained.* The homeowner *receives* security information via a control panel, the PC, or a browser, collectively called an interface. The interface *displays* prompting messages and system status information on the control panel, the PC, or the browser window. Homeowner interaction takes the following form . . .

Referring to the grammatical parse, verbs are *SafeHome* processes and can be represented as bubbles in a subsequent DFD. Nouns are either external entities (boxes), data or control objects

(arrows), or data stores (double lines). Nouns and verbs can be associated with one another (e.g., each sensor is assigned a number and type; therefore number and type are attributes of the data object **sensor**).

Therefore, by performing a grammatical parse on the processing narrative for a bubble at any DFD level, you can generate much useful information about how to proceed with the refinement to the next level. Using this information, a level 1 DFD is shown in Figure 7.2. The context level process shown in Figure 7.1 has been expanded into six processes derived from an examination of the grammatical parse. Similarly, the information flow between processes at level 1 has been derived from the parse. In addition, information flow continuity is maintained between levels 0 and 1.

The processes represented at DFD level 1 can be further refined into lower levels. For example, the process *monitor sensors* can be refined into a level 2 DFD as shown in Figure 7.3. Note once again that information flow continuity has been maintained between levels.

The refinement of DFDs continues until each bubble performs a simple function. That is, until the process represented by the bubble performs a function that would be easily implemented as a program component. In Chapter 8, I discuss a concept, called *cohesion*, that can be used to assess the processing focus of a given function. For now, we strive to refine DFDs until each bubble is "single-minded."

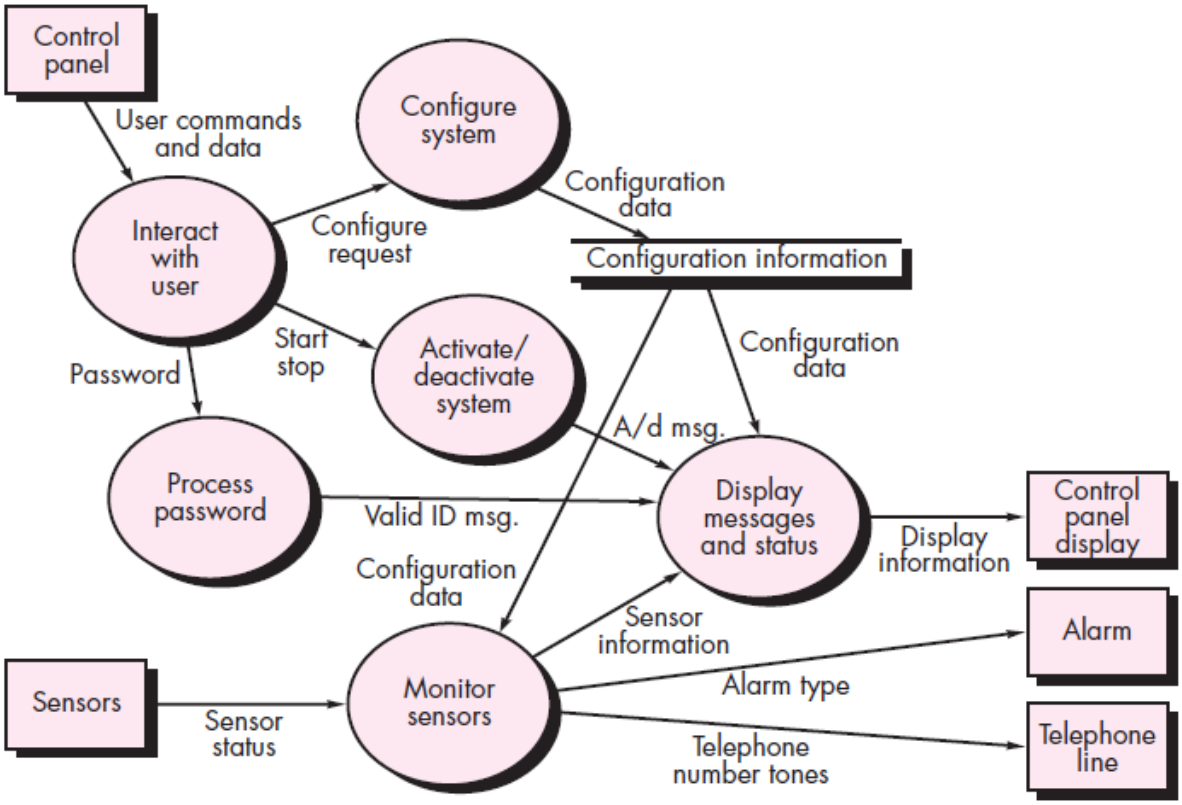


Fig 2.2. Level 1 DFD for *SafeHome* security function.

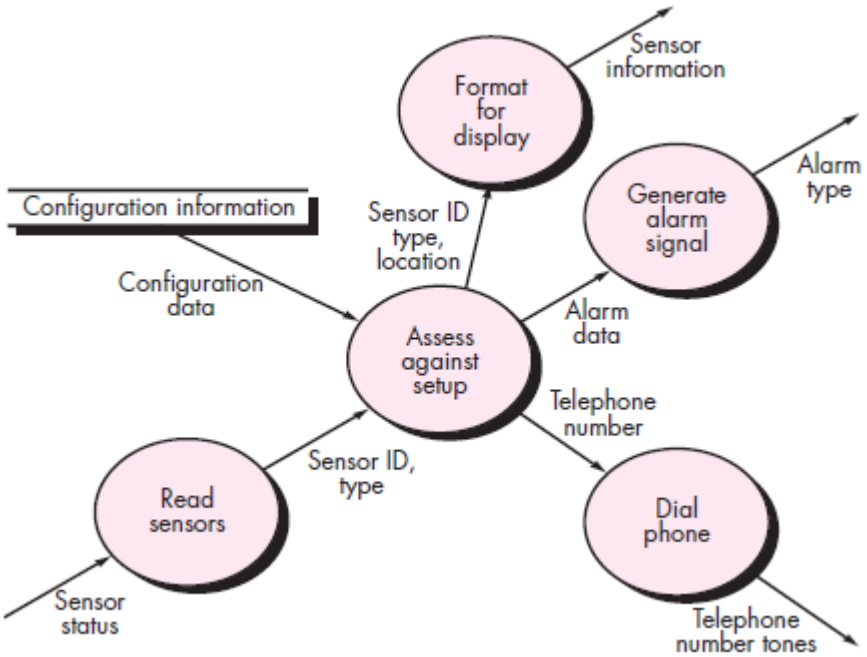


Fig 2.3. Level 2 DFD that refines the *monitor sensors* process

Creating a Control Flow Model

For some types of applications, the data model and the data flow diagram are all that is necessary to obtain meaningful insight into software requirements. As I have already noted, however, a large class of applications are "driven" by events rather than data, produce control information rather than reports or displays, and process information with heavy concern for time and performance. Such applications require the use of *control flow modeling* in addition to data flow modeling.

I have already noted that an event or control item is implemented as a Boolean value (e.g., true or false, on or off, 1 or 0) or a discrete list of conditions (e.g., empty, jammed, full). To select potential candidate events, the following guidelines are suggested:

• List all sensors that are "read" by the software.

• List all interrupt conditions.

• List all "switches" that are actuated by an operator.

• List all data conditions.

• Recalling the noun/verb parse that was applied to the processing narrative, review all "control items" as possible control specification inputs/outputs.

• Describe the behavior of a system by identifying its states, identify how each state is reached, and define the transitions between states. • Focus on possible omissions—a very common error in specifying control; for example, ask: "Is there any other way I can get to this state or exit from it?"

Among the many events and control items that are part of *SafeHome* software are **sensor event** (i.e., a sensor has been tripped), **blink flag** (a signal to blink the display), and **start/stop switch** (a signal to turn the system on or off).

THE CONTROL SPECIFICATION

A *control specification* (CSPEC) represents the behavior of the system (at the level from which it has been referenced) in two different ways.3 The CSPEC contains a state diagram that is a sequential specification of behavior. It can also contain a program activation table—a combinatorial specification of behavior.

Figure 2.4 depicts a preliminary state diagram4 for the level 1 control flow model for *SafeHome.* The diagram indicates how the system responds to events as it traverses the four states defined at this level. By reviewing the state diagram, you can determine the behavior of the system and, more important, ascertain whether there are "holes" in the specified behavior.

For example, the state diagram (Figure 7.4) indicates that the transitions from the **Idle** state can occur if the system is reset, activated, or powered off. If the system is

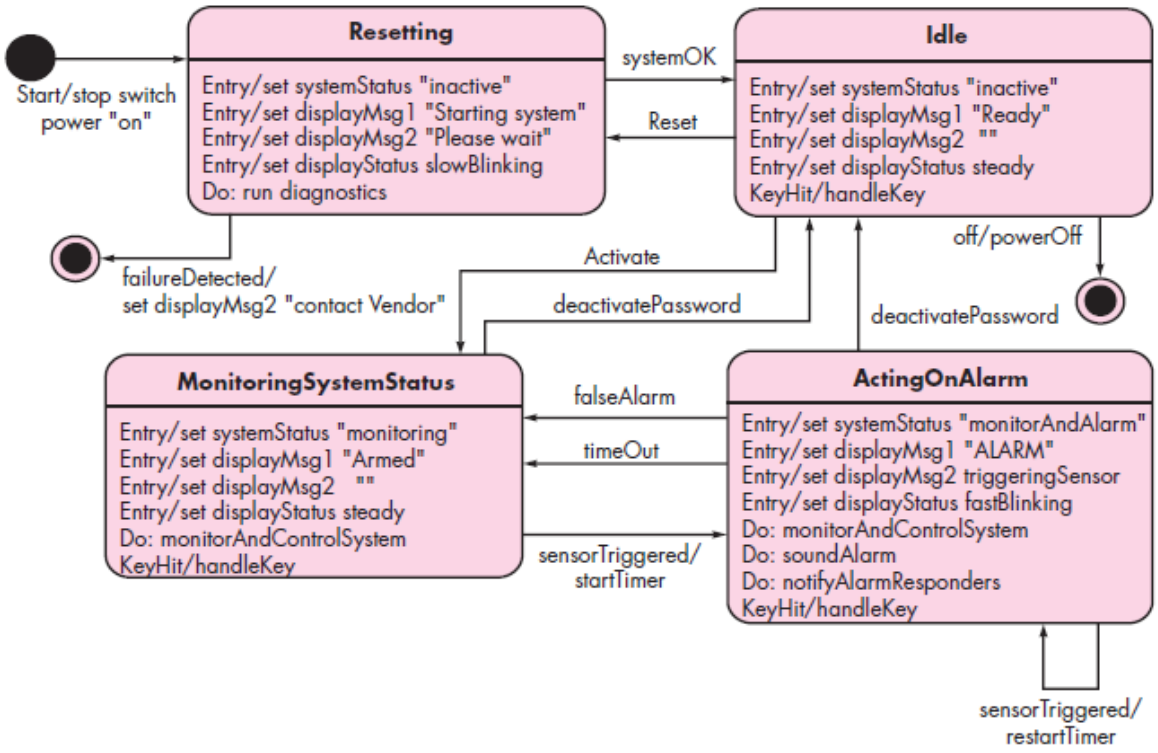


Fig 2.4. State diagram for *SafeHome* security function

activated (i.e., alarm system is turned on), a transition to the **Monitoring- SystemStatus** state occurs, display messages are changed as shown, and the process *monitorAndControlSystem* is invoked. Two transitions occur out of the **MonitoringSystemStatus** state—(1) when the system is deactivated, a transition occurs back to the **Idle** state; (2) when a sensor is triggered into the **ActingOnAlarm** state. All transitions and the content of all states are considered during the review.

A somewhat different mode of behavioral representation is the process activation table. The PAT represents information contained in the state diagram in the context of processes, not states. That is, the table indicates which processes (bubbles) in the flow model will be invoked when an event occurs. The PAT can be used as a guide for a designer who must build an executive that controls the processes represented at this level. A PAT for the level 1 flow model of *SafeHome* software is shown in Figure 2.5.

The CSPEC describes the behavior of the system, but it gives us no information about the inner working of the processes that are activated as a result of this behavior.

THE PROCESS SPECIFICATION

The *process specification* (PSPEC) is used to describe all flow model processes that appear at the final level of refinement. The content of the process specification can

input events						
sensor event	0	0	0	0	1	0
blink flag	0	0	1	1	0	0
start stop switch	0	1	0	0	0	0
display action status complete	0	0	0	1	0	0
in-progress	0	0	1	0	0	0
time out	0	0	0	0	0	1
output						
alarm signal	0	0	0	0	1	0
process activation						
monitor and control system	0	1	0	0	1	1
activate/deactivate system	0	1	0	0	0	0
display messages and status	1	0	1	1	1	1
interact with user	1	0	0	1	0	1

Fig 2.5. Process activation table for *SafeHome* security function

include narrative text, a program design language (PDL) description5 of the process algorithm, mathematical equations, tables, or UML activity diagrams. By providing a PSPEC to accompany each bubble in the flow model, you can create a "mini-spec" that serves as a guide for design of the software component that will implement the bubble. To illustrate the use of the PSPEC, consider the *process password* transform represented in the flow model for *SafeHome* (Figure 7.2). The PSPEC for this function might take the form:

PSPEC: process password (at control panel). The *process password* transform performs password validation at the control panel for the *SafeHome* security function. *Process password* receives a four-digit password from the *interact with user* function. The password is first compared to the master password stored within the system. If the master password matches, <valid id message = true> is passed to the *message and status display* function. If the master password does not match, the four digits are compared to a table of secondary passwords (these may be assigned to house guests and/or workers who require entry to the home when the owner is not present). If the password matches an entry within the table, <valid id message = true> is

passed to the *message and status display function*. If there is no match, <valid id message = false> is passed to the message and status display function. If additional algorithmic detail is desired at this stage, a program design language representation may also be included as part of the PSPEC. However, many believe that the PDL version should be postponed until component design commences.

CREATING A BEHAVIORAL MODEL.

The modeling notation represents static elements of the requirements model. It is now time to make a transition to the dynamic behavior of the system or product. To accomplish this, you can represent the behavior of the system as a function of specific events and time.

The *behavioral model* indicates how software will respond to external events or stimuli. To create the model, you should perform the following steps:

1. Evaluate all use cases to fully understand the sequence of interaction within the system.

2. Identify events that drive the interaction sequence and understand how these events relate to specific objects.

3. Create a sequence for each use case.

4. Build a state diagram for the system.

5. Review the behavioral model to verify accuracy and consistency.

Each of these steps is discussed in the sections that follow.

Identifying Events with the Use Case

In Chapter 6 you learned that the use case represents a sequence of activities that involves actors and the system. In general, an event occurs whenever the system and an actor exchange information. In Section 7.2.3, I indicated that an event is *not* the information that has been exchanged, but rather the fact that information has been exchanged.

A use case is examined for points of information exchange. To illustrate, we reconsider the use case for a portion of the *SafeHome* security function.

The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.

The underlined portions of the use case scenario indicate events. An actor should be identified for each event; the information that is exchanged should be noted, and any conditions or constraints should be listed.

As an example of a typical event, consider the underlined use case phrase "homeowner uses the keypad to key in a four-digit password." In the context of the requirements model, the object, **Homeowner,**7 transmits an event to the object **ControlPanel.** The event might be called *password entered*.

The information transferred is the four digits that constitute the password, but this is not an essential part of the behavioral model. It is important to note that some events have an explicit impact on the flow of control of the use case, while others have no direct impact on the flow of control. For example, the event *password entered* does not explicitly change the flow of control of the use case, but the results of the event *password compared* (derived from the interaction "password is compared with the valid password stored in the system") will have an explicit impact on the information and control flow of the *SafeHome* software.

Once all events have been identified, they are allocated to the objects involved. Objects can be responsible for generating events (e.g., **Homeowner** generates the *password entered* event) or recognizing events that have occurred elsewhere (e.g., **ControlPanel** recognizes the binary result of the *password compared* event).

State Representations

In the context of behavioral modeling, two different characterizations of states must be considered: (1) the state of each class as the system performs its function and (2) the state of the system as observed from the outside as the system performs its function.8

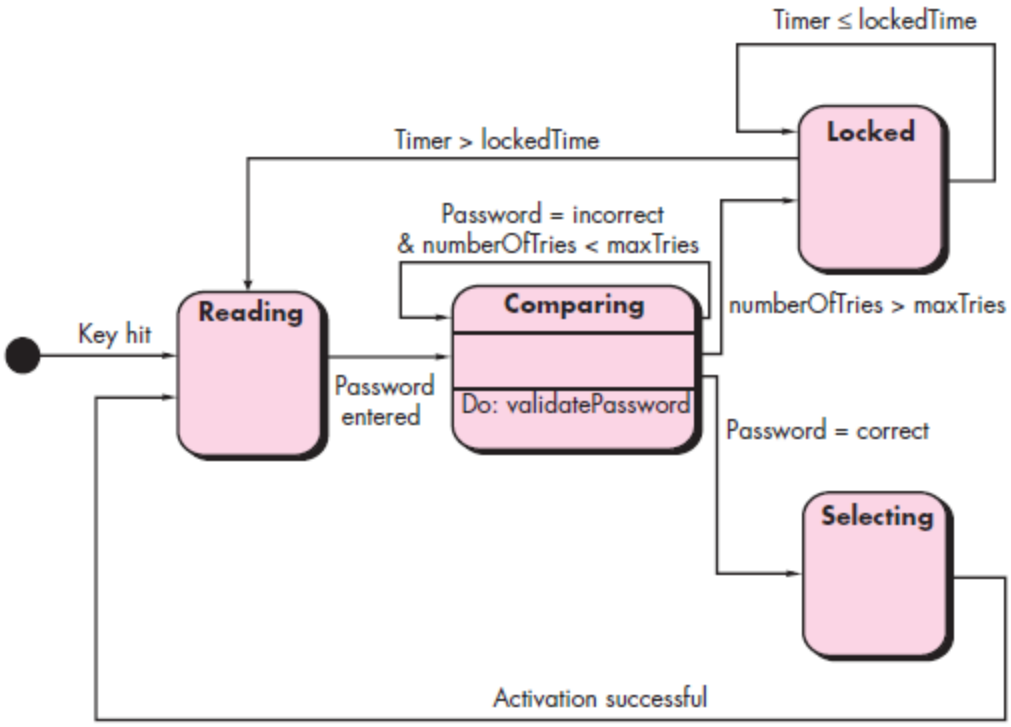
The state of a class takes on both passive and active characteristics [Cha93]. A *passive state* is simply the current status of all of an object's attributes. For example, the passive state of the class **Player** (in the video game application discussed in Chapter 6) would include the current position and orientation attributes of **Player** as well as other features of **Player** that are relevant to the game (e.g., an attribute that indicates magic wishes remaining).

The *active state* of an object indicates the current status of the object as it undergoes a continuing transformation or processing. The class **Player** might have the following active states: *moving, at rest, injured, being cured; trapped, lost,* and so forth. An event (sometimes called a *trigger*) must occur to force an object to make a transition from one active state to another.

Two different behavioral representations are discussed in the paragraphs that follow. The first indicates how an individual class changes state based on external events and the second shows the behavior of the software as a function of time. **State diagrams for analysis classes.**

One component of a behavioral model is a UML state diagram9 that represents active states for each class and the events (triggers) that cause changes between these active states. Figure 7.6 illustrates a state diagram for the **ControlPanel** object in the *SafeHome* security function.

Each arrow shown in Figure 7.6 represents a transition from one active state of an object to another. The labels shown for each arrow represent the event that



State diagram for the ControlPanel Class

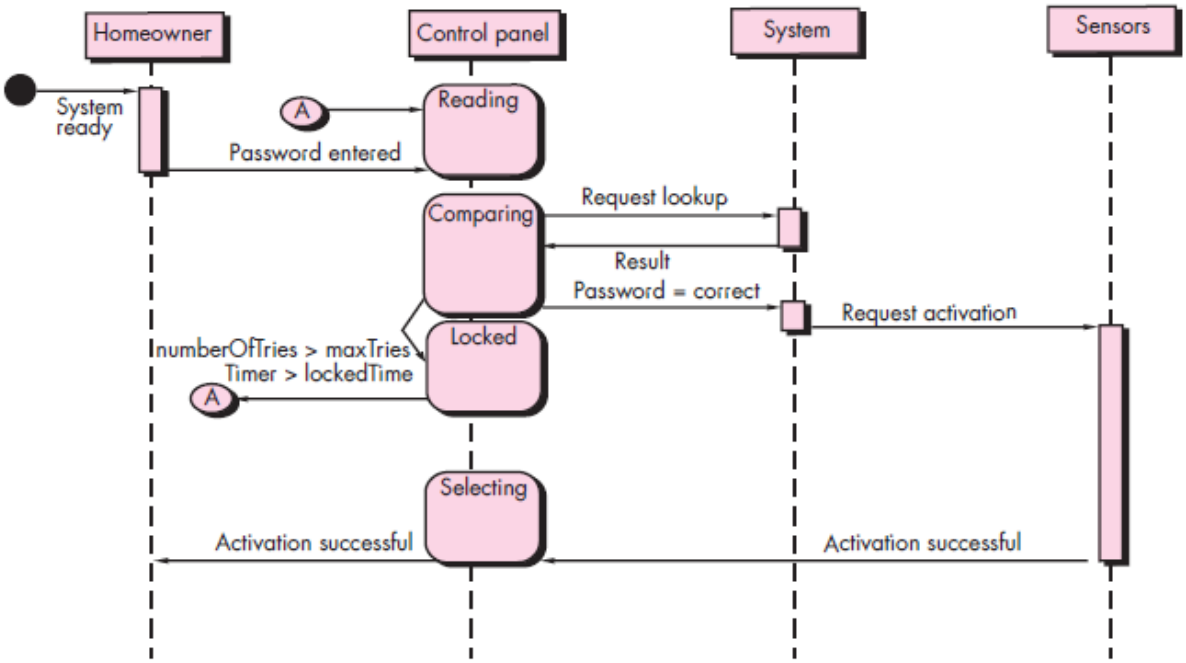
triggers the transition. Although the active state model provides useful insight into the "life history" of an object, it is possible to specify additional information to provide more depth in understanding the behavior of an object. In addition to specifying the event that causes the transition to occur, you can specify a guard and an action [Cha93]. A *guard* is a Boolean condition that must be satisfied in order for the transition to occur. For example, the guard for the transition from the "reading" state to the "comparing" state in Figure 7.6 can be determined by examining the use case: if (password input _ 4 digits) then compare to stored password

In general, the guard for a transition usually depends upon the value of one or more attributes of an object. In other words, the guard depends on the passive state of the object.

An *action* occurs concurrently with the state transition or as a consequence of it and generally involves one or more operations (responsibilities) of the object. For example, the action connected to the *password entered* event (Figure 7.6) is an operation named *validatePassword()* that accesses a **password** object and performs a digit-by-digit comparison to validate the entered password.

Sequence diagrams.

The second type of behavioral representation, called a *sequence diagram* in UML, indicates how events cause transitions from object to object. Once events have been identified by examining a use case, the modeler



Sequence diagram (partial) for the *SafeHome* security function creates a sequence diagram—a representation of how events cause flow from one object to another as a function of time. In essence, the sequence diagram is a shorthand version of the use case. It represents key classes and the events that cause behavior to flow from class to class.

Figure 7.7 illustrates a partial sequence diagram for the *SafeHome* security function. Each of the arrows represents an event (derived from a use case) and indicates how the event channels behavior between *SafeHome* objects. Time is measured vertically (downward), and the narrow vertical rectangles represent time spent in processing an activity. States may be shown along a vertical time line.

The first event, *system ready*, is derived from the external environment and channels behavior to the **Homeowner** object. The homeowner enters a password. A *request lookup* event is passed to **System,** which looks up the password in a simple database and returns a *result* (*found* or *not found*) to **ControlPanel** (now in the *comparing* state). A valid password results in a *password=correct* event to **System,** which activates **Sensors** with a *request activation* event. Ultimately, control is passed back to the homeowner with the *activation successful* event.

Once a complete sequence diagram has been developed, all of the events that cause transitions between system objects can be collated into a set of input events and output events (from an object). This information is useful in the creation of an effective design for the system to be built.

POSSIBLE QUESTIONS

PART – B

1. Explain Structure Analysis Model?
2. Describe about the Requirement Analysis.
3. Explain in detail about data modeling concepts with examples.
4. Describe flow-oriented modeling with examples.
5. Describe about Process Specification and Control Specification.
6. Develop state diagram and sequence diagram that could serve as a basis for understanding the requirements for a SafeHome Security function.
7. Elucidate the steps to create a behavioral model.

UNIT-III

SYLLABUS

Design Engineering: Design with the Context of Software Engineering-Design Process and Design Quality-Design Concepts-Creating An Architectural Design: Software Architecture-Data Design-Architectural Design- Assessing Alternative Architectural Designs-Mapping Data Flow into Software Architecture.

DESIGN ENGINEERING:

Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. Design principles establish an overriding philosophy that guides you in the design work you must perform. Design concepts must be understood before the mechanics of design practice are applied, and design practice itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

Design is pivotal to successful software engineering. In the early 1990s Mitch Kapor, the creator of Lotus 1-2-3, presented a "software design manifesto" in *Dr. Dobbs Journal.* He said:

What is design? It's where you stand with a foot in two worlds—the world of technology and the world of people and human purposes—and you try to bring the two together. . . .

The Roman architecture critic Vitruvius advanced the notion that well-designed buildings were those which exhibited firmness, commodity, and delight. The same might be said of good software.

Firmness: A program should not have any bugs that inhibit its function.

Commodity: A program should be suitable for the purposes for which it was intended.

Delight: The experience of using the program should be a pleasurable one. Here we have the beginnings of a theory of design for software.

The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight.

To accomplish this, you must practice diversification and then convergence. Belady states that "diversification is the acquisition of a repertoire of alternatives, the raw material of design: components, component solutions, and knowledge, all contained in catalogs, textbooks, and the mind." Once this diverse set of information is assembled, you must pick and choose

elements from the repertoire that meet the requirements defined by requirements engineering and the analysis model. As this occurs, alternatives are considered and rejected and you converge on "one particular configuration of components, and thus the creation of the final product".

Diversification and convergence combine intuition and judgment based on experience in building similar entities, a set of principles and/or heuristics that guide the way in which the model evolves, a set of criteria that enables quality to be judged, and a process of iteration that ultimately leads to a final design representation.

Software design changes continually as new methods, better analysis, and broader understanding evolve. Even today, most software design methodologies lack the depth, flexibility, and quantitative nature that are normally associated with more classical engineering design disciplines. However, methods for software design do exist, criteria for design quality are available, and design notation can be applied.

DESIGN WITH THE CONTEXT OF SOFTWARE ENGINEERING-

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and modeled, software design is the last software engineering action within the modeling activity and sets the stage for **construction** (code generation and testing).

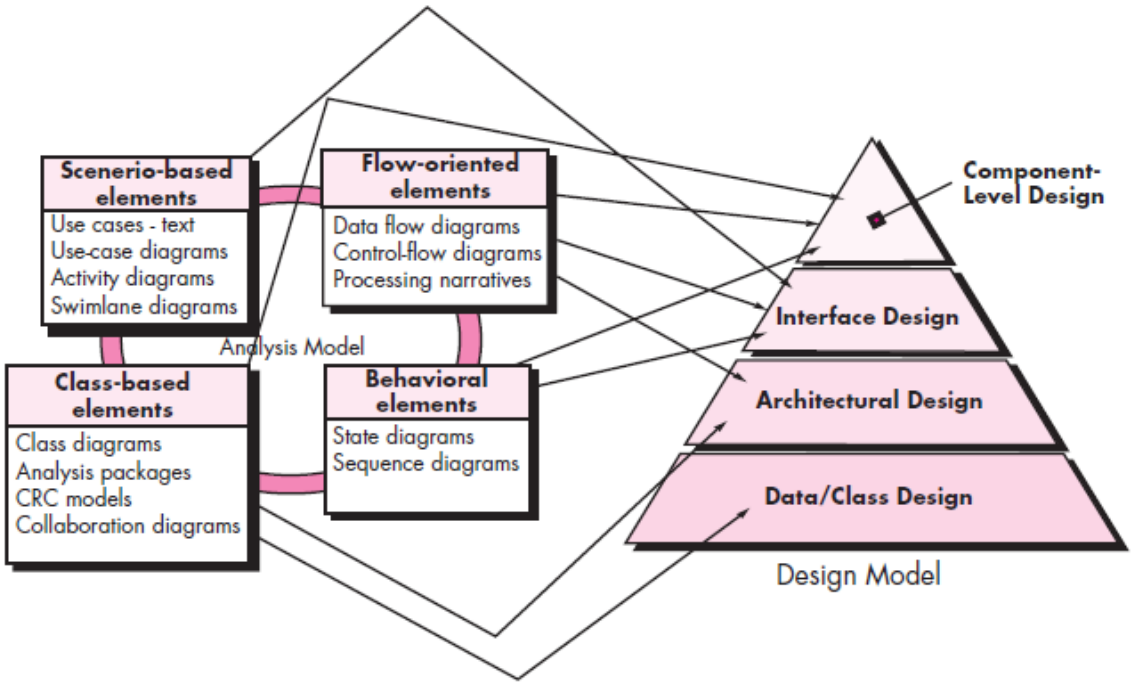


Fig. Translating the requirements model into the design model

Each of the elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in Figure. The requirements model, manifested by scenario-based, class-based, flow-oriented, and behavioral elements, feed the design task. Using design notation and design methods discussed in later chapters, design produces a data/class design, an architectural design, an interface design, and a component design.

Data/class design

The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action. Part of class design may occur in conjunction with the design of software architecture. More detailed class design occurs as each software component is designed.

Architectural design

The architectural design defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented. The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

Interface design

The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

Component-level design

The component-level design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

During design you make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained. But why is design so important?

The importance of software design can be stated with a single word—*quality*.

Design is the place where quality is fostered in software engineering. Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow. Without design, you risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction— a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle.

DESIGN QUALITY

Software Quality Guidelines and Attributes

Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews discussed. McGlaughlin suggests three characteristics that serve as a guide for the evaluation of a good design:

• The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.

• The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.

• The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Each of these characteristics is actually a goal of the design process. But how is each of these goals achieved?

Quality Guidelines.

In order to evaluate the quality of a design representation, you and other members of the software team must establish technical criteria for good design. In Section 8.3, I discuss design concepts that also serve as software quality criteria. For the time being, consider the following guidelines:

1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (3) can be implemented in an evolutionary fashion,2 thereby facilitating implementation and testing.

2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.

3. A design should contain distinct representations of data, architecture, interfaces, and components.

4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

5. A design should lead to components that exhibit independent functional characteristics.

6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.

7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

8. A design should be represented using a notation that effectively communicates its meaning.

These design guidelines are not achieved by chance. They are achieved through the application of fundamental design principles, systematic methodology, and thorough review.

Quality Attributes.

Hewlett-Packard developed a set of software quality attributes that has been given the acronym FURPS—functionality, usability, reliability, performance, and supportability. The FURPS quality attributes represent a target for all software design:

• *Functionality* is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.

• *Usability* is assessed by considering human factors, overall aesthetics, consistency, and documentation.

• *Reliability* is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.

• *Performance* is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.

• *Supportability* combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, *maintainability*—and in addition, testability, compatibility, configurability, the ease with which a system can be installed, and the ease with which problems can be localized.

Not every software quality attribute is weighted equally as the software design is developed. One application may stress functionality with a special emphasis on security.

Another may demand performance with particular emphasis on processing speed. A third might focus on reliability. Regardless of the weighting, it is important to note that these quality attributes must be considered as design commences, *not* after the design is complete and construction has begun.

The Evolution of Software Design

The evolution of software design is a continuing process that has now spanned almost six decades.

Early design work concentrated on criteria for the development of modular programs and methods for refining software structures in a topdown manner.

Procedural aspects of design definition evolved into a philosophy called *structured programming*.

Later work proposed methods for the translation of data flow or data structure into a design definition.

Newer design approaches proposed an object-oriented approach to design derivation.

More recent emphasis in software design has been on software architecture and the design patterns that can be used to implement software architectures and lower levels of design abstractions.

Growing emphasis on aspect-oriented methods model-driven development, and test-driven development emphasize techniques for achieving more effective modularity and architectural structure in the designs that are created.

A number of design methods, growing out of the work just noted, are being applied throughout the industry. Like the analysis methods, each software design method introduces unique heuristics and notation, as well as a somewhat parochial view of what characterizes design quality. Yet, all of these methods have a number of common characteristics:

(1) a mechanism for the translation of the requirements model into a design representation,

(2) a notation for representing functional components and their interfaces,

(3) heuristics for refinement and partitioning, and

(4) guidelines for quality assessment.

Regardless of the design method that is used, you should apply a set of basic concepts to data, architectural, interface, and component-level design. These concepts are considered in the sections that follow.

DESIGN CONCEPTS

A set of fundamental software design concepts has evolved over the history of software engineering. Although the degree of interest in each concept has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps you answer the following questions:

• What criteria can be used to partition software into individual components?

• How is function or data structure detail separated from a conceptual representation of the software?

• What uniform criteria define the technical quality of a software design?

M. A. Jackson once said: "The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work, and getting it right." Fundamental software design concepts provide the necessary framework for "getting it right."

In the sections that follow, I present a brief overview of important software design concepts that span both traditional and object-oriented software development.

1. Abstraction

When you consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

As different levels of abstraction are developed, you work to create both procedural and data abstractions. A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).5

A *data abstraction* is a named collection of data that describes a data object. In the context of the procedural abstraction *open*, we can define a data abstraction called **door.** Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door.**

2. Architecture

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system" [Sha95a]. In its simplest form, architecture is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components. In a broader sense, however, components can be generalized to represent major system elements and their interactions. One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to solve common design problems.

Shaw and Garlan describe a set of properties that should be specified as part of an architectural design:

Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models. *Structural models* represent architecture as an organized collection of program components.

Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications. *Dynamic models* address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events. *Process models* focus on the design of the business or technical process that the system must accommodate. Finally, *functional models* can be used to represent the functional hierarchy of a system.

A number of different *architectural description languages* (ADLs) have been developed to represent these models. Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another.

Some researchers argue that the derivation of software architecture should be separated from design and occurs between requirements engineering actions and more conventional design actions. Others believe that the derivation of architecture is an integral part of the design process.

3. Patterns

Brad Appleton defines a *design pattern* in the following manner: "A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns" [App00]. Stated in another way, a design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine

(1) whether the pattern is applicable to the current work,

(2) whether the pattern can be reused (hence, saving design time), and

(3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

4. Separation of Concerns

Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A *concern* is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

For two problems, *p*1 and *p*2, if the perceived complexity of *p*1 is greater than the perceived complexity of *p*2, it follows that the effort required to solve *p*1 is greater than the effort required to solve *p*2. As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem.

It also follows that the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately. This leads to a divide-and-conquer strategy—it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to software modularity.

Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement. Each will be discussed in the subsections that follow.

5. Modularity

Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called *modules*, that are integrated to satisfy problem requirements.

It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.

The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

Recalling my discussion of separation of concerns, it is possible to conclude that if you subdivide software indefinitely the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure, the effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the

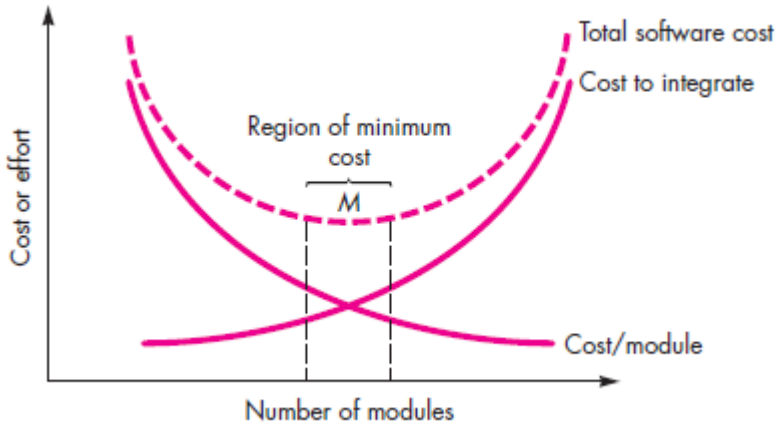


Fig .Modularity and software cost

same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, *M*, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict *M* with assurance.

The curves shown in Figure do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of *M*. Undermodularity or overmodularity should be avoided. But how do you know the vicinity of *M*? How modular should you make software? The answers to these questions require an understanding of other design concepts considered later in this chapter.

You modularize a design (and the resulting program) so that development can be more easily planned; software increments can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

6. Information Hiding

The concept of modularity leads you to a fundamental question: "How do I decompose a software solution to obtain the best set of modules?" The principle of information hiding suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

7. Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. In landmark papers on software design, refinement techniques that enhance module independence.

Functional independence is achieved by developing modules with "singleminded" function and an "aversion" to excessive interaction with other modules. Stated another way, you should design software so that each module addresses a specific subset of requirements and has a simple

interface when viewed from other parts of the program structure. It is fair to ask why independence is important.

Software with effective modularity, that is, independent modules, is easier to develop because function can be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria: cohesion and coupling.

Cohesion is an indication of the relative functional strength of a module. *Coupling* is an indication of the relative interdependence among modules.

Cohesion is a natural extension of the information-hiding concept. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions. However, "schizophrenic" components (modules that perform many unrelated functions) are to be avoided if a good design is to be achieved.

Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, you should strive for the lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagate throughout a system.

8. Refinement

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth [Wir71]. A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

Refinement is actually a process of *elaboration.* You begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal

workings of the function or the internal structure of the information. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses. Both concepts allow you to create a complete design model as the design evolves.

9. Aspects

As requirements analysis occurs, a set of "concerns" is uncovered. These concerns "include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts" [AOS07]. Ideally, a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independently. In practice, however, some of these concerns span the entire system and cannot be easily compartmentalized.

As design begins, requirements are refined into a modular design representation. Consider two requirements, *A* and *B*. Requirement *A crosscuts* requirement *B* "if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account" [Ros04].

For example, consider two requirements for the **SafeHomeAssured.com** WebApp. Requirement *A* is described via the **ACS-DCV** use case discussed. A design refinement would focus on those modules that would enable a registered user to access video from cameras placed throughout a space. Requirement *B* is a generic security requirement that states that *a registered user must be validated prior to using*

SafeHomeAssured.com. This requirement is applicable for all functions that are available to registered *SafeHome* users. As design refinement occurs, *A** is a design representation for requirement *A* and *B** is a design representation for requirement *B*. Therefore, *A** and *B** are representations of concerns, and *B* crosscuts A**. An *aspect* is a representation of a crosscutting concern. Therefore, the design representation, *B**, of the requirement *a registered user must be validated prior to using* **SafeHomeAssured.com,** is an aspect of the *SafeHome* WebApp. It is important to identify aspects so that the design can properly accommodate them as refinement and modularization occur. In an ideal context, an aspect is implemented as a separate module (component) rather than as software fragments that are "scattered" or "tangled" throughout many components [Ban06]. To accomplish this, the design architecture should support a mechanism for defining an aspect—a module that enables the concern to be implemented across all other concerns that it crosscuts.

10. Refactoring

An important design activity suggested for many agile methods, *refactoring* is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior. Fowler [Fow00] defines refactoring in the following manner: "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."

When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. For example, a first design iteration might yield a component that exhibits low cohesion (i.e., it performs three functions that have only limited relationship to one another). After careful consideration, you may decide that the component should be refactored into three separate components, each exhibiting high cohesion. The result will be software that is easier to integrate, easier to test, and easier to maintain.

11. Object-Oriented Design Concepts

The object-oriented (OO) paradigm is widely used in modern software engineering. Appendix 2 has been provided for those readers who may be unfamiliar with OO design concepts such as classes and objects, inheritance, messages, and polymorphism, among others.

12. Design Classes

The requirements model defines a set of analysis classes (Chapter 6). Each describes some element of the problem domain, focusing on aspects of the problem that are user visible. The level of abstraction of an analysis class is relatively high.

As the design model evolves, you will define a set of *design classes* that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution. Five different types of design classes, each representing a different layer of the design architecture, can be developed [Amb01]:

• *User interface classes* define all abstractions that are necessary for humancomputer interaction (HCI). In many cases, HCI occurs within the context of a *metaphor* (e.g., a checkbook, an order form, a fax machine), and the design classes for the interface may be visual representations of the elements of the metaphor.

• *Business domain classes* are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.

• *Process classes* implement lower-level business abstractions required to fully manage the business domain classes.

• *Persistent classes* represent data stores (e.g., a database) that will persist beyond the execution of the software.

• *System classes* implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

As the architecture forms, the level of abstraction is reduced as each analysis class is transformed into a design representation. That is, analysis classes represent data objects (and associated services that are applied to them) using the jargon of the business domain. Design classes present significantly more technical detail as a guide for implementation.

Arlow and Neustadt [Arl02] suggest that each design class be reviewed to ensure that it is "well-formed." They define four characteristics of a well-formed design class:

Complete and sufficient. A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class.

For example, the class **Scene** defined for video-editing software is complete only if it contains all attributes and methods that can reasonably be associated with the creation of a video scene. Sufficiency ensures that the design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

Primitiveness. Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing. For example, the class **VideoClip** for video-editing software might have attributes start-point and end-point to indicate the start and end points of the clip (note that the raw video loaded into the system may be longer than the clip that is used). The methods, *setStartPoint()* and *setEndPoint()*, provide the only means for establishing start and end points for the clip.

High cohesion. A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities. For example, the class **VideoClip** might contain a set of methods for editing the video clip. As long as each method focuses solely on attributes associated with the video clip, cohesion is maintained.

Low coupling. Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled (all design classes collaborate with all other design classes), the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes. This restriction, called the *Law of Demeter* [Lie03], suggests that a method should only send messages to methods in neighboring classes.6

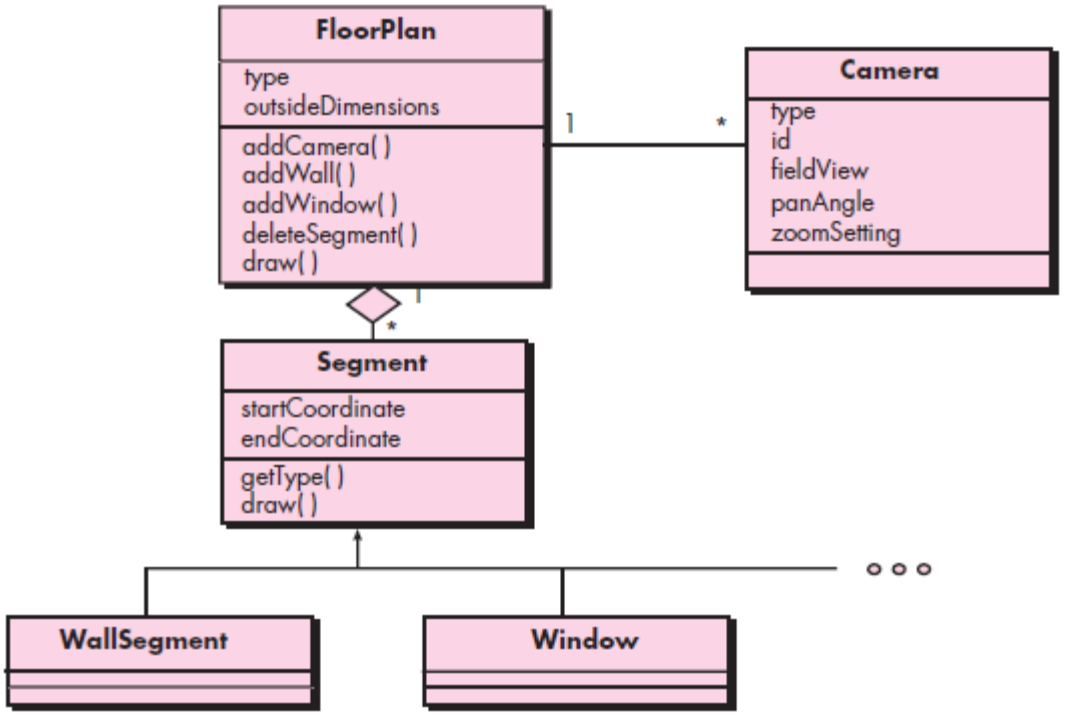


Fig. Design class for FloorPlan and composite aggregation for the class (see sidebar discussion)

CREATING AN ARCHITECTURAL DESIGN:

Software Architecture

In their landmark book on the subject, Shaw and Garlan [Sha96] discuss software architecture in the following manner:

Ever since the first program was divided into modules, software systems have had architectures, and programmers have been responsible for the interactions among the modules and the global properties of the assemblage. Historically, architectures have been implicit—accidents of implementation, or legacy systems of the past. Good software developers have often adopted one

or several architectural patterns as strategies for system organization, but they use these patterns informally and have no means to make them explicit in the resulting system.

Today, effective software architecture and its explicit representation and design have become dominant themes in software engineering.

What Is Architecture?

When you consider the architecture of a building, many different attributes come to mind. At the most simplistic level, you think about the overall shape of the physical structure. But in reality, architecture is much more. It is the manner in which the various components of the building are integrated to form a cohesive whole. It is the way in which the building fits into its environment and meshes with other buildings in its vicinity. It is the degree to which the building meets its stated purpose and satisfies the needs of its owner. It is the aesthetic feel of the structure—the visual impact of the building—and the way textures, colors, and materials are combined to create the external facade and the internal "living environment." It is small details— the design of lighting fixtures, the type of flooring, the placement of wall hangings, the list is almost endless. And finally, it is art.

But architecture is also something else. It is "thousands of decisions, both big and small" [Tyr05]. Some of these decisions are made early in design and can have a profound impact on all other design actions. Others are delayed until later, thereby eliminating overly restrictive constraints that would lead to a poor implementation of the architectural style.

But what about software architecture? Bass, Clements, and Kazman [Bas03] define this elusive term in the following way:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

The architecture is not the operational software. Rather, it is a representation that enables you to

(1) analyze the effectiveness of the design in meeting its stated requirements,

(2) consider architectural alternatives at a stage when making design changes is still relatively easy, and

(3) reduce the risks associated with the construction of the software.

This definition emphasizes the role of "software components" in any architectural representation. In the context of architectural design, a software component can be something as simple as a program module or an object-oriented class, but it can also be extended to include databases and "middleware" that enable the configuration of a network of clients and servers. The properties of components are those characteristics that are necessary for an understanding of how the components interact with other components. At the architectural level, internal properties (e.g., details of an algorithm) are not specified. The relationships between components can be as simple as a procedure call from one module to another or as complex as a database access protocol.

Some members of the software engineering community (e.g., [Kaz03]) make a distinction between the actions associated with the derivation of a software architecture (what I call "architectural design") and the actions that are applied to derive the software design. As one reviewer of this edition noted:

There is a distinct difference between the terms architecture and design. A *design* is an instance of an *architecture* similar to an object being an instance of a class. For example, consider the client-server architecture. I can design a network-centric software system in many different ways from this architecture using either the Java platform (Java EE) or Microsoft platform (.NET framework). So, there is one architecture, but many designs can be created based on that architecture. Therefore, you cannot mix "architecture" and "design" with each other.

Although I agree that a software design is an instance of a specific software architecture, the elements and structures that are defined as part of an architecture are the root of every design that evolves from them. Design begins with a consideration of architecture.

In this book the design of software architecture considers two levels of the design pyramid (Figure 8.1)—data design and architectural design. In the context of the preceding discussion, data design enables you to represent the data component of the architecture in conventional systems and class definitions (encompassing attributes and operations) in object-oriented systems. Architectural design focuses on the representation of the structure of software components, their properties, and interactions.

Why Is Architecture Important?

In a book dedicated to software architecture, Bass and his colleagues [Bas03] identify three key reasons that software architecture is important:

• Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.

• The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.

• Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together" [Bas03]. The architectural design model and the architectural patterns contained within it are transferable. That is, architecture genres, styles, and patterns (Sections 9.2 through 9.4) can be applied to the design of other systems and represent a set of abstractions that enable software engineers to describe architecture in predictable ways.

Architectural Descriptions

Each of us has a mental image of what the word *architecture* means. In reality, however, it means different things to different people. The implication is that different stakeholders will see an architecture from different viewpoints that are driven by different sets of concerns. This implies that an architectural description is actually a set of work products that reflect different views of the system. For example, the architect of a major office building must work with a variety of different stakeholders.

The primary concern of the owner of the building (one stakeholder) is to ensure that it is aesthetically pleasing and that it provides sufficient office space and infrastructure to ensure its profitability. Therefore, the architect must develop a description using views of the building that address the owner's concerns. The viewpoints used are a three-dimensional drawings of the building (to illustrate the aesthetic view) and a set of two-dimensional floor plans to address this stakeholder's concern for office space and infrastructure.

But the office building has many other stakeholders, including the structural steel fabricator who will provide steel for the building skeleton. The structural steel fabricator needs detailed architectural information about the structural steel that will support the building, including types of I-beams, their dimensions, connectivity, materials, and many other details. These concerns are addressed by different work products that represent different views of the architecture. Specialized drawings (another viewpoint) of the structural steel skeleton of the building focus on only one of many of the fabricator's concerns.

An architectural description of a software-based system must exhibit characteristics that are analogous to those noted for the office building. Tyree and Akerman [Tyr05] note this when they write: "Developers want clear, decisive guidance on how to proceed with design. Customers

want a clear understanding on the environmental changes that must occur and assurances that the architecture will meet their business needs. Other architects want a clear, salient understanding of the architecture's key aspects." Each of these "wants" is reflected in a different view represented using a different viewpoint.

The IEEE Computer Society has proposed IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*, [IEE00], with the following objectives: (1) to establish a conceptual framework and vocabulary for use during the design of software architecture, (2) to provide detailed guidelines for representing an architectural description, and (3) to encourage sound architectural design practices.

The IEEE standard defines an *architectural description* (AD) as "a collection of products to document an architecture." The description itself is represented using multiple views, where each *view* is "a representation of a whole system from the perpective of a related set of [stakeholder] concerns." A *view* is created according to rules and conventions defined in a *viewpoint*—"a specification of the conventions for constructing and using a view" [IEE00]. A number of different work products that are used to develop different views of the software architecture are discussed later in this chapter.

Architectural Decisions

Each view developed as part of an architectural description addresses a specific stakeholder concern. To develop each view (and the architectural description as a whole) the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern. Therefore, architectural decisions themselves can be considered to be one view of the architecture.

The reasons that decisions were made provide insight into the structure of a system and its conformance to stakeholder concerns.

As a system architect, you can use the template suggested in the sidebar to document each major decision. By doing this, you provide a rationale for your work and establish an historical record that can be useful when design modifications must be made.

DATA DESIGN

DATA DESIGN ELEMENTS

Like other software engineering activities, data design (sometimes referred to as *data architecting*) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.

The structure of data has always been an important part of software design. At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications. At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system. At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself. In every case, data design plays an important role.

ARCHITECTURAL DESIGN

As architectural design begins, the software to be developed must be put into context—that is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction. This information can generally be acquired from the requirements model and all other information gathered during requirements engineering. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes.

An *archetype* is an abstraction (similar to a class) that represents one element of system behavior. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail. Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived. In the sections that follow we examine each of these architectural design tasks in a bit more detail.

Representing the System in Context

At the architectural design level, a software architect uses an *architectural context diagram* (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in Figure 3.5.

Referring to the figure, systems that interoperate with the *target system* (the system for which an architectural design is to be developed) are represented as

• *Superordinate systems*—those systems that use the target system as part of some higher-level processing scheme.

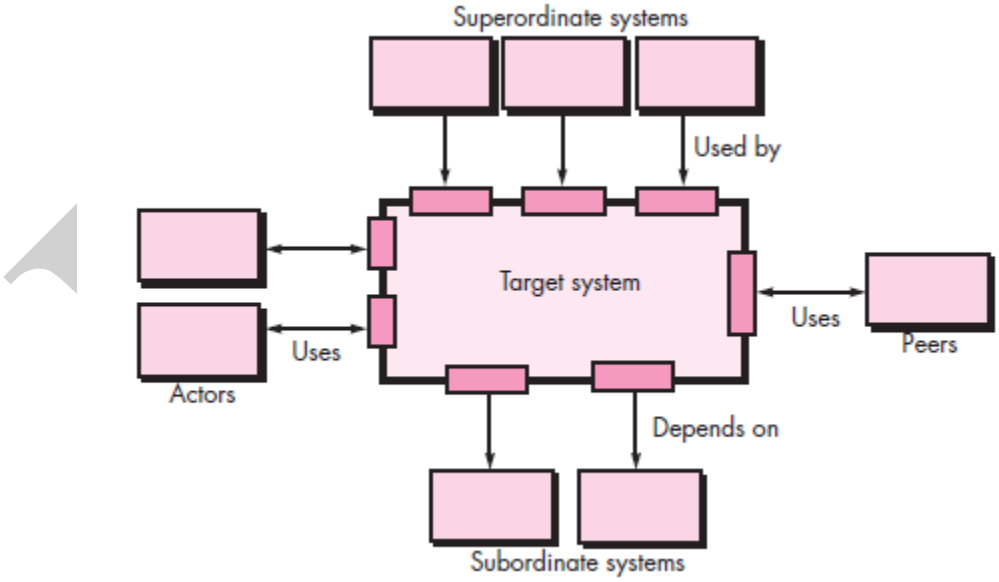
• *Subordinate systems*—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.

• *Peer-level systems*—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.

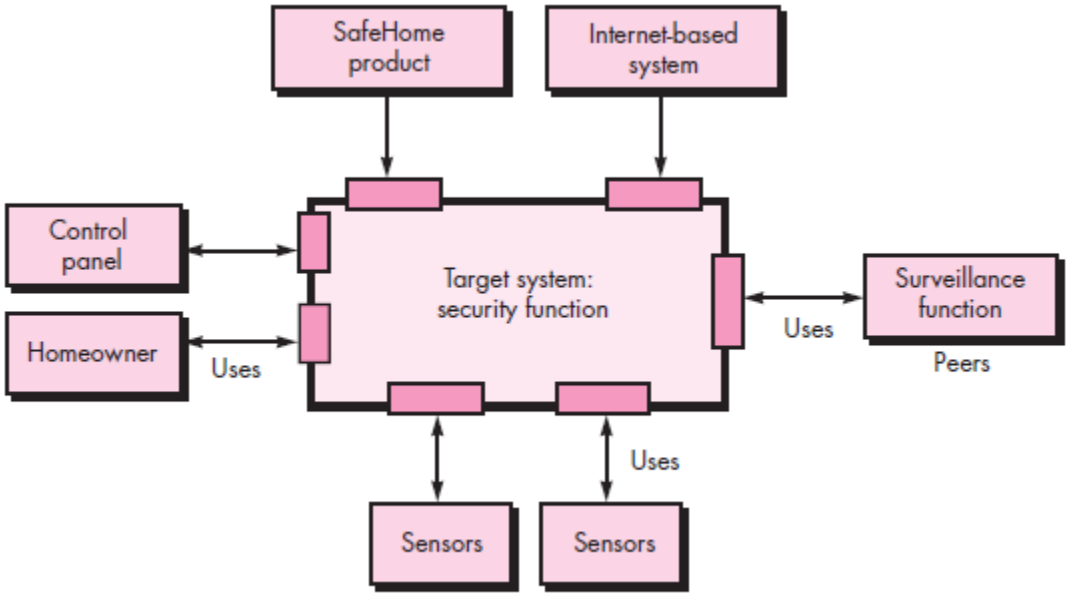
• *Actors*—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

To illustrate the use of the ACD, consider the home security function of the *SafeHome* product. The overall *SafeHome* product controller and the Internet-based system are both superordinate to the security function and are shown above the



3.5.Architectural context diagram



Architectural context diagram for the *SafeHome* security function

function in Figure. The surveillance function is a *peer system* and uses (is used by) the home security function in later versions of the product. The homeowner and control panels are actors that are both producers and consumers of information used/produced by the security software. Finally, sensors are used by the security software and are shown as subordinate to it.

As part of the architectural design, the details of each interface shown in Figure would have to be specified. All data that flow into and out of the target system must be identified at this stage.

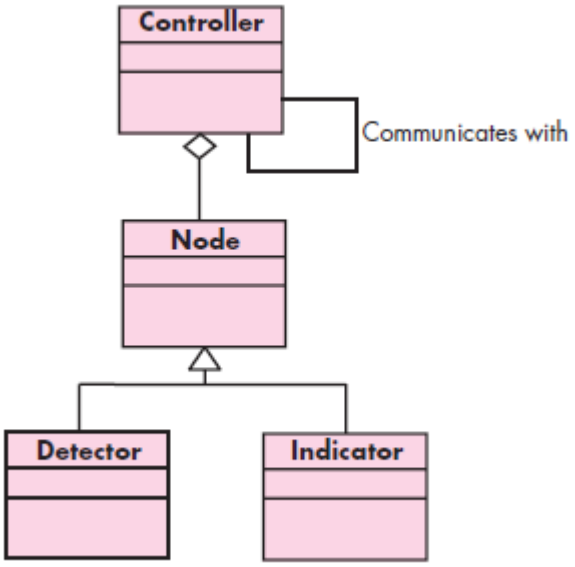
Defining Archetypes

An *archetype* is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

In many cases, archetypes can be derived by examining the analysis classes defined as part of the requirements model. Continuing the discussion of the *SafeHome* home security function, you might define the following archetypes:

• **Node.** Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors and (2) a variety of alarm (output) indicators.

• **Detector.** An abstraction that encompasses all sensing equipment that feeds information into the target system.



UML relationships for *SafeHome* security function archetypes

• **Indicator.** An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.

• **Controller.** An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another. Each of these archetypes is depicted using UML notation as shown in Figure 9.7. Recall that the archetypes form the basis for the architecture but are abstractions that must be further refined as architectural design proceeds. For example, **Detector** might be refined into a class hierarchy of sensors.

Refining the Architecture into Components

As the software architecture is refined into components, the structure of the system begins to emerge. But how are these components chosen? In order to answer this question, you begin with the classes that were described as part of the requirements model.

These analysis classes represent entities within the application (business) domain that must be addressed within the software architecture. Hence, the application domain is one source for the

derivation and refinement of components. Another source is the infrastructure domain. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain. For example, memory management components, communication components, database components, and task management components are often integrated into the software architecture.

The interfaces depicted in the architecture context diagram (Section 9.4.1) imply one or more specialized components that process the data that flows across the interface. In some cases (e.g., a graphical user interface), a complete subsystem architecture with many components must be designed.

Continuing the *SafeHome* home security function example, you might define the set of top-level components that address the following functionality:

• *External communication management*—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.

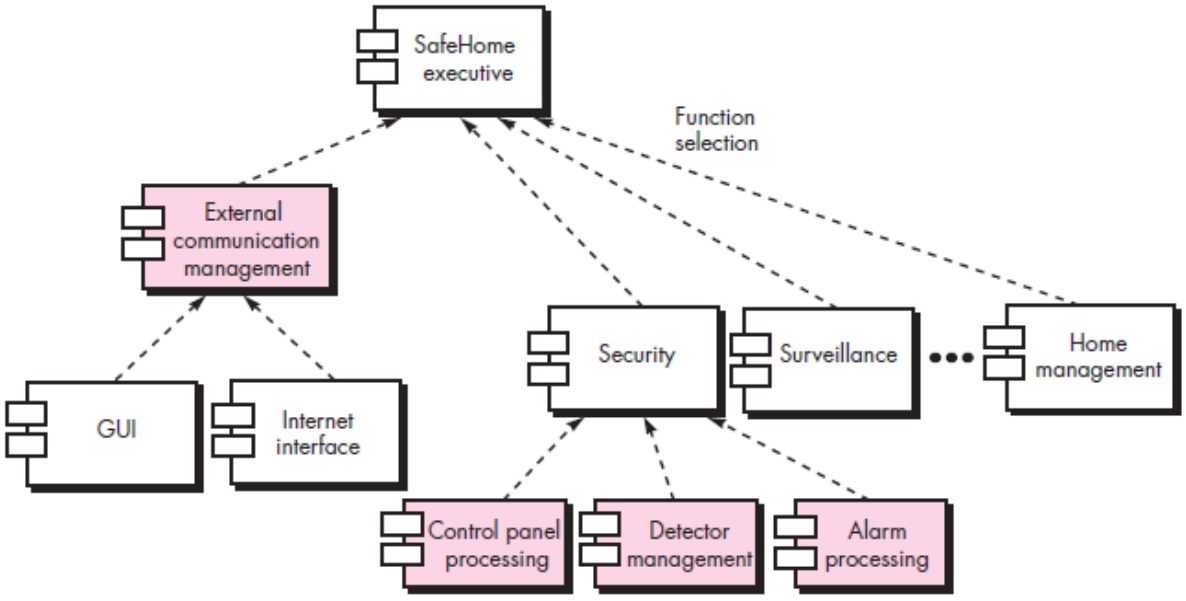
• *Control panel processing*—manages all control panel functionality.

• *Detector management*—coordinates access to all detectors attached to the system.

• *Alarm processing*—verifies and acts on all alarm conditions.

Each of these top-level components would have to be elaborated iteratively and then positioned within the overall *SafeHome* architecture. Design classes (with appropriate attributes and operations) would be defined for each. It is important to note, however, that the design details of all attributes and operations would not be specified until component-level design.

The overall architectural structure (represented as a UML component diagram) is illustrated in Figure. Transactions are acquired by *external communication management* as they move in from components that process the *SafeHome* GUI and the



Overall architectural structure for *SafeHome* with top-level components

Internet interface. This information is managed by a *SafeHome* executive component that selects the appropriate product function (in this case security). The *control panel processing* component interacts with the homeowner to arm/disarm the security function. The *detector management* component polls sensors to detect an alarm condition, and the *alarm processing* component produces output when an alarm is detected.

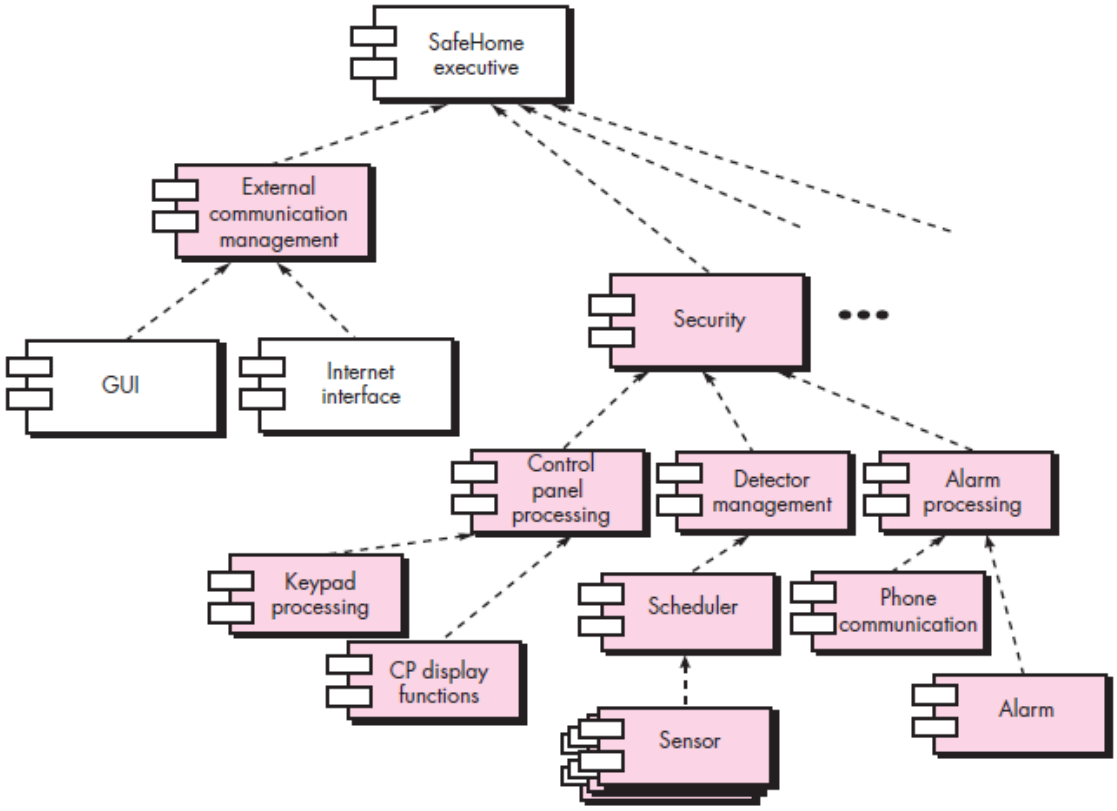
Describing Instantiations of the System

The architectural design that has been modeled to this point is still relatively high level. The context of the system has been represented, archetypes that indicate the important abstractions within the problem domain have been defined, the overall structure of the system is apparent, and the major software components have been identified. However, further refinement (recall that all design is iterative) is still necessary.

To accomplish this, an actual instantiation of the architecture is developed. By this I mean that the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.

Figure illustrates an instantiation of the *SafeHome* architecture for the security system. Components shown in Figure 9.8 are elaborated to show additional detail. For example, the *detector management* component interacts with a *scheduler* infrastructure component that

implements polling of each *sensor* object used by the security system. Similar elaboration is performed for each of the components represented in Figure 9.8.



An instantiation of the security function with component elaboration

ASSESSING ALTERNATIVE ARCHITECTURAL DESIGNS

In their book on the evaluation of software architectures, Clements and his colleagues [Cle03] state:

To put it bluntly, an architecture is a bet, a wager on the success of a system. Wouldn't it be nice to know in advance if you've placed your bet on a winner, as opposed to waiting until the system is mostly completed before knowing whether it will meet its requirements or not? If you're buying a system or paying for its development, wouldn't you like to have some assurance that it's started off down the right path? If you're the architect yourself, wouldn't you like to have a good way to validate your intuitions and experience, so that you can sleep at night knowing that the trust placed in your design is well founded?

Indeed, answers to these questions would have value. Design results in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved. In the sections that follow, I present two different approaches for the assessment of alternative architectural designs. The first method uses an iterative method to assess design trade-offs. The second approach applies a pseudo-quantitative technique for assessing design quality.

An Architecture Trade-Off Analysis Method

The Software Engineering Institute (SEI) has developed an *architecture trade-off analysis method* (ATAM) [Kaz98] that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively:

1. *Collect scenarios.* A set of use cases (Chapters 5 and 6) is developed to represent the system from the user's point of view.

2. *Elicit requirements, constraints, and environment description.* This information is determined as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed.

3. *Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements.* The architectural style(s) should be described using one of the following architectural views:

• *Module view* for analysis of work assignments with components and the degree to which information hiding has been achieved.

• *Process view* for analysis of system performance.

• *Data flow view* for analysis of the degree to which the architecture meets functional requirements.

4. *Evaluate quality attributes by considering each attribute in isolation.* The number of quality attributes chosen for analysis is a function of the time available for review and the degree to which quality attributes are relevant to the system at hand. Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.

5. *Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.* This can be accomplished by making small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed *sensitivity points.*

6. *Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.* The SEI describes this approach in the following manner [Kaz98]:

Once the architectural sensitivity points have been determined, finding trade-off points is simply the identification of architectural elements to which multiple attributes are sensitive. For example, the performance of a client-server architecture might be highly sensitive to the number of servers (performance increases, within some range, by increasing the number of servers). . . . The number of servers, then, is a trade-off point with respect to this architecture.

These six steps represent the first ATAM iteration. Based on the results of steps 5 and 6, some architecture alternatives may be eliminated, one or more of the remaining architectures may be modified and represented in more detail, and then the ATAM steps are reapplied.6

Architectural Complexity

A useful technique for assessing the overall complexity of a proposed architecture is to consider dependencies between components within the architecture. These dependencies are driven by information/control flow within the system. Zhao [Zha98] suggests three types of dependencies:

Sharing dependencies represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers. For example, for two components **u** and **v**, if **u** and **v** refer to the same global data, then there exists a shared dependence relationship between **u** and **v**.

Flow dependencies represent dependence relationships between producers and consumers of resources. For example, for two components **u** and **v,** if **u** must complete before control flows into **v** (prerequisite), or if **u** communicates with **v** by parameters, then there exists a flow dependence relationship between **u** and **v.**

Constrained dependencies represent constraints on the relative flow of control among a set of activities. For example, for two components **u** and **v**, **u** and **v** cannot execute at the same time (mutual exclusion), then there exists a constrained dependence relationship between **u** and **v.**

The sharing and flow dependencies noted by Zhao are similar to the concept of coupling discussed in Chapter 8. Coupling is an important design concept that is applicable at the architectural level and at the component level. Simple metrics for evaluating coupling are discussed in Chapter 23.

9.5.3 Architectural Description Languages

The architect of a house has a set of standardized tools and notation that allow the design to be represented in an unambiguous, understandable fashion. Although the software architect can draw on UML notation, other diagrammatic forms, and a few related tools, there is a need for a more formal approach to the specification of an architectural design.

Architectural description language (ADL) provides a semantics and syntax for describing a software architecture. Hofmann and his colleagues [Hof01] suggest that an ADL should provide the designer with the ability to decompose architectural components, compose individual components into larger architectural blocks, and represent interfaces (connection mechanisms) between components. Once descriptive, languagebased techniques for architectural design have been established, it is more likely that effective assessment methods for architectures will be established as the design evolves.

MAPPING DATA FLOW INTO SOFTWARE ARCHITECTURE

The architectural styles discussed in Section 9.3.1 represent radically different architectures. So it should come as no surprise that a comprehensive mapping that accomplishes the transition from the requirements model to a variety of architectural styles does not exist. In fact, there is no practical mapping for some architectural styles, and the designer must approach the translation of requirements to design for these styles in using the techniques discussed in Section 9.4.

To illustrate one approach to architectural mapping, consider the call and return architecture—an extremely common structure for many types of systems. The call and return architecture can reside within other more sophisticated architectures discussed earlier in this chapter. For example, the architecture of one or more components of a client-server architecture might be call and return.

A mapping technique, called *structured design* is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram to software architecture. The transition from information flow (represented as a DFD) to program structure is accomplished as part of a sixstep process:

(1) the type of information flow is established,

(2) flow boundaries are indicated,

(3) the DFD is mapped into the program structure,

(4) control hierarchy is defined,

(5) the resultant structure is refined using design measures and heuristics, and

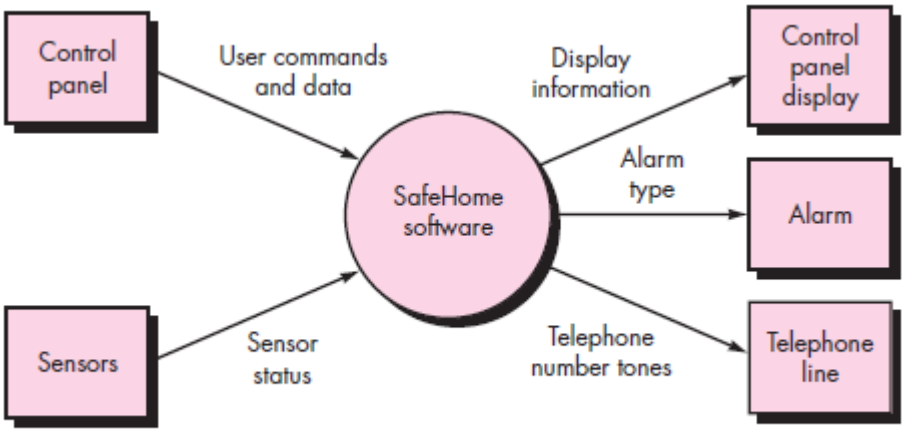
(6) the architectural description is refined and elaborated.

As a brief example of data flow mapping, I present a step-by-step "transform" mapping for a small part of the *SafeHome* security function. In order to perform the mapping, the type of information flow must be determined. One type of information flow is called *transform flow* and exhibits a linear quality. Data flows into the system along an *incoming flow path* where it is transformed from an external world representation into internalized form. Once it has been internalized, it is processed at a *transform center*. Finally, it flows out of the system along an *outgoing flow path* that transforms the data into external world form.9

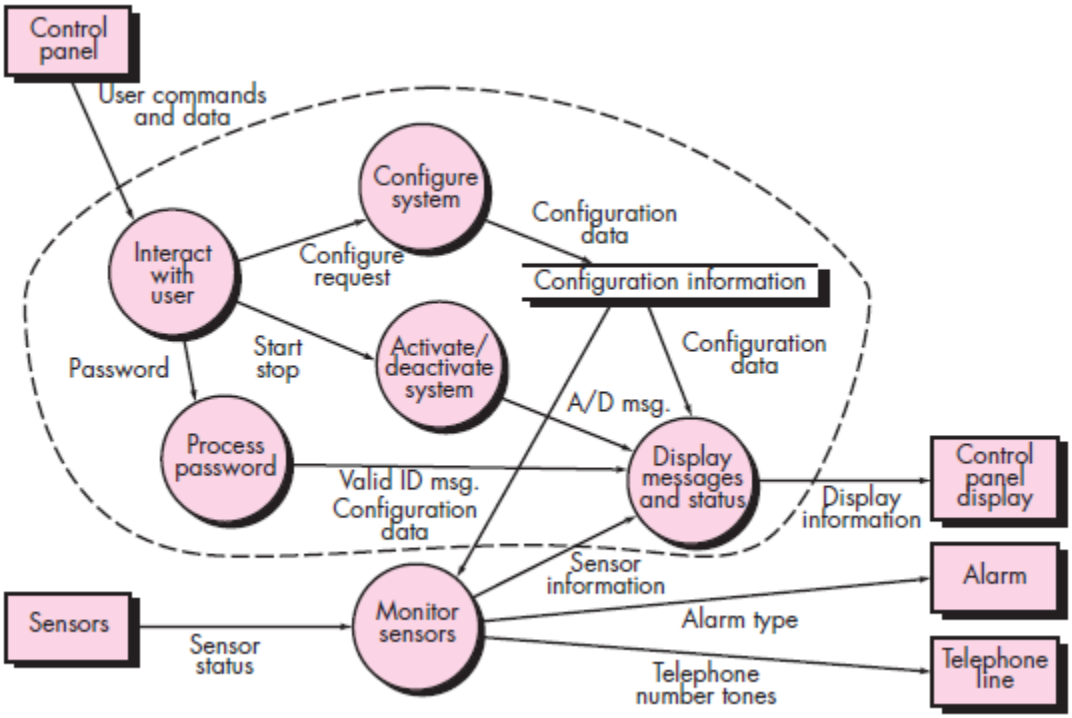
Transform Mapping

Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style. To illustrate this approach, we again consider the *SafeHome* security function.10 One element of the analysis model is a set of data flow diagrams that describe information flow within the security function. To map these data flow diagrams into a software architecture, you would initiate the following design steps:

Step 1. Review the fundamental system model. The fundamental system model or context diagram depicts the security function as a single transformation, representing the external producers and consumers of data that flow into and out of the function. Figure 9.10 depicts a level 0 context model, and Figure 9.11 shows refined data flow for the security function.



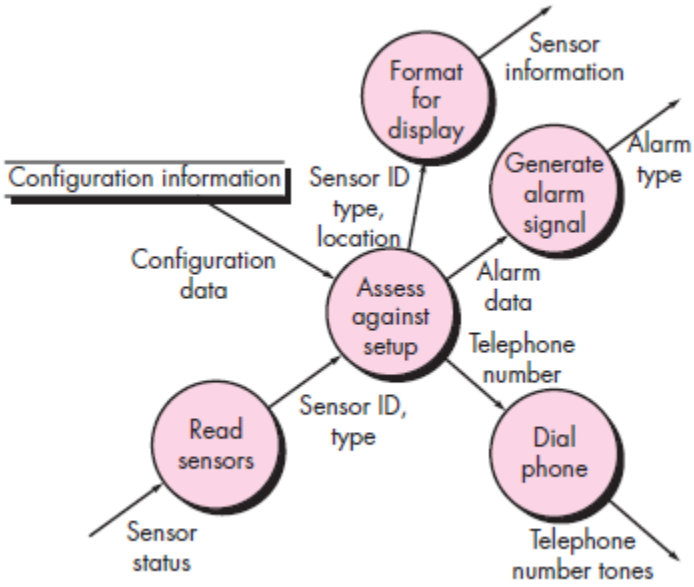
Context-level DFD for the *SafeHome* security function



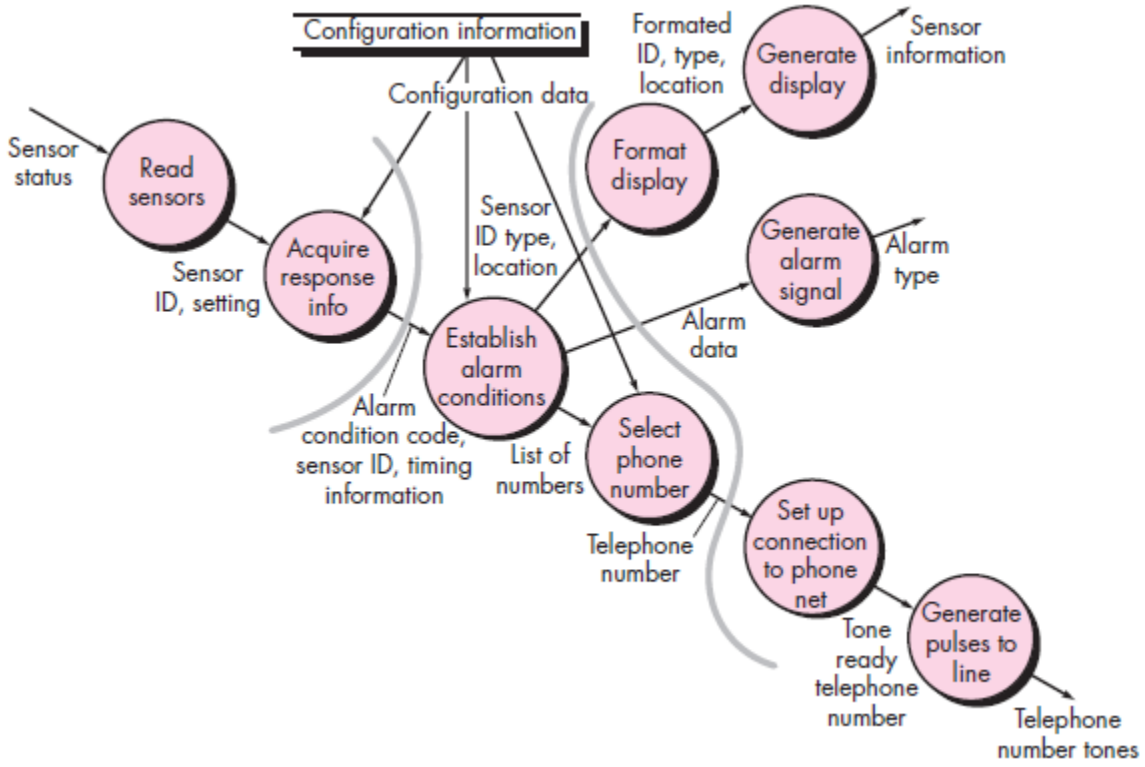
Level 1 DFD for the *SafeHome* security function

Step 2. Review and refine data flow diagrams for the software. Information obtained from the requirements model is refined to produce greater detail. For example, the level 2 DFD for

monitor sensors is examined, and a level 3 data flow diagram is derived as shown in Figure. At level 3, each transform in



Level 2 DFD that refines the *monitor sensors* transform



Level 3 DFD for *monitor sensors* with flow boundaries

the data flow diagram exhibits relatively high cohesion. That is, the process implied by a transform performs a single, distinct function that can be implemented as a component in the *SafeHome* software. Therefore, the DFD in Figure contains sufficient detail for a "first cut" at the design of architecture for the *monitor sensors* subsystem, and we proceed without further refinement.

Step 3. Determine whether the DFD has transform or transaction flow11 characteristics. Evaluating the DFD we see data entering the software along one incoming path and exiting along three outgoing paths. Therefore, an overall transform characteristic will be assumed for information flow.

Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries. Incoming data flows along a path in which information is converted from external to internal form; outgoing flow converts internalized data to external form. Incoming and outgoing flow boundaries are open to interpretation.

That is, different designers may select slightly different points in the flow as boundary locations. In fact, alternative design solutions can be derived by varying the placement of flow boundaries. Although care should be taken when boundaries are selected, a variance of one bubble along a flow path will generally have little impact on the final program structure.

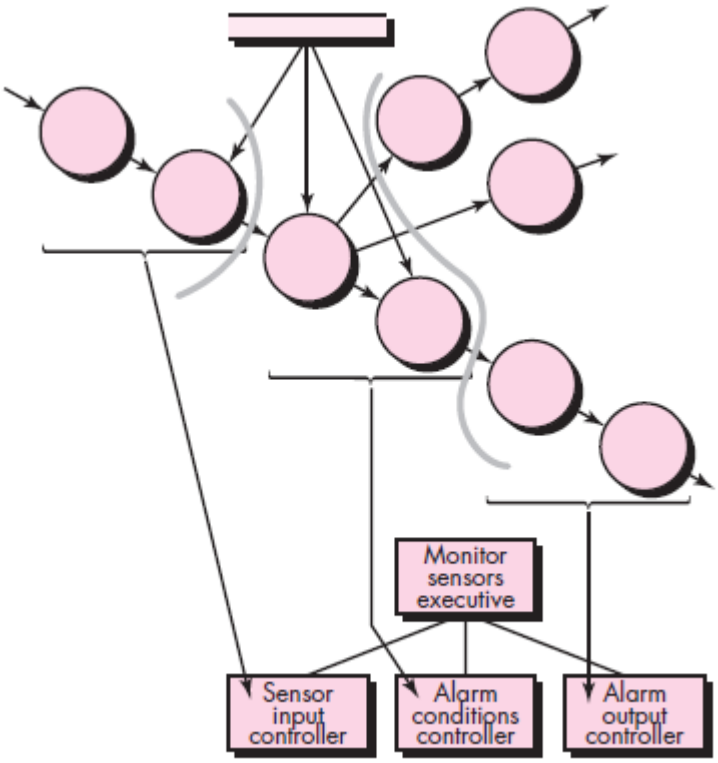
Flow boundaries for the example are illustrated as shaded curves running vertically through the flow in Figure 9.13. The transforms (bubbles) that constitute the transform center lie within the two shaded boundaries that run from top to bottom in the figure. An argument can be made to readjust a boundary (e.g., an incoming flow boundary separating *read sensors* and *acquire response info* could be proposed).

The emphasis in this design step should be on selecting reasonable boundaries, rather than lengthy iteration on placement of divisions.

Step 5. Perform "first-level factoring." The program architecture derived using this mapping results in a top-down distribution of control. *Factoring* leads to a program structure in which top-level components perform decision making and lowlevel components perform most input, computation, and output work. Middle-level components perform some control and do moderate amounts of work.

When transform flow is encountered, a DFD is mapped to a specific structure (a call and return architecture) that provides control for incoming, transform, and outgoing information processing. This first-level factoring for the *monitor sensors* subsystem is illustrated in Figure 9.14. A main controller (called *monitor sensors executive*) resides at the top of the program structure and coordinates the following subordinate control functions:

• An incoming information processing controller, called *sensor input controller*, coordinates receipt of all incoming data.



First-level factoring for *monitor sensors*

• A transform flow controller, called *alarm conditions controller,* supervises all operations on data in internalized form (e.g., a module that invokes various data transformation procedures).

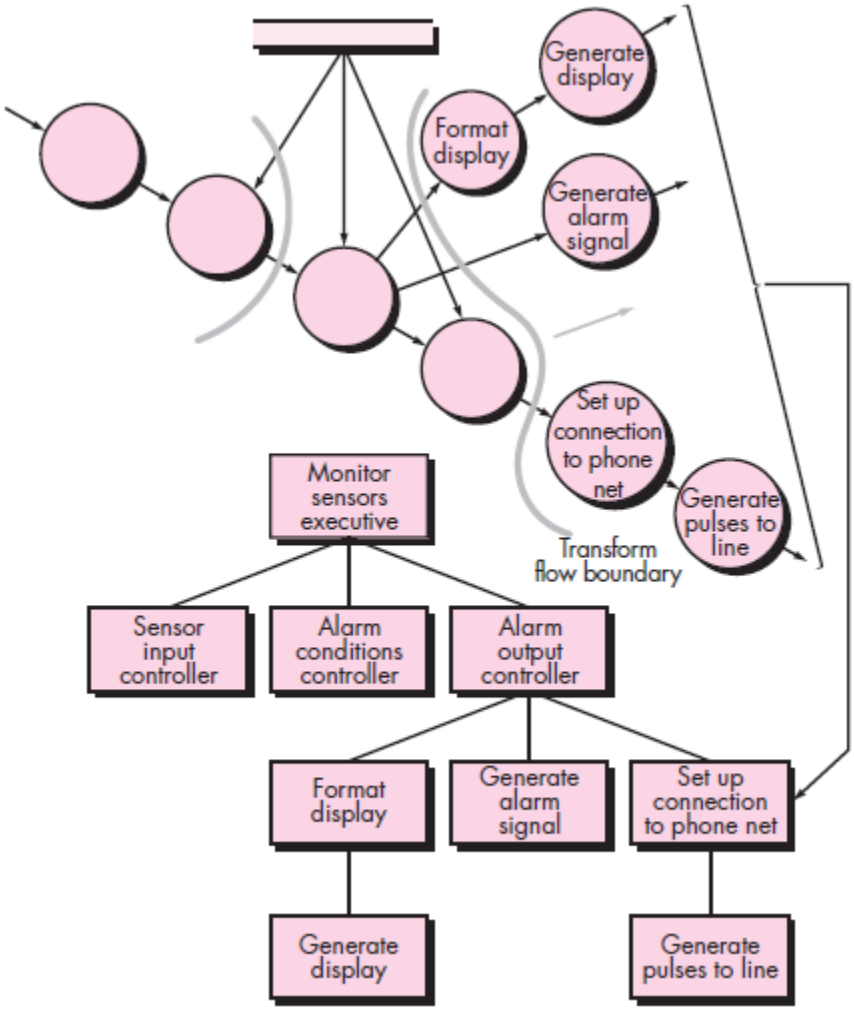
• An outgoing information processing controller, called *alarm output controller,* coordinates production of output information.

Although a three-pronged structure is implied by Figure 9.14, complex flows in large systems may dictate two or more control modules for each of the generic control functions described previously. The number of modules at the first level should be limited to the minimum that can accomplish control functions and still maintain good functional independence characteristics.

Step 6. Perform "second-level factoring." Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at the transform center boundary and moving outward along incoming and then

outgoing paths, transforms are mapped into subordinate levels of the software structure. The general approach to secondlevel factoring is illustrated in Figure.

Although Figure illustrates a one-to-one mapping between DFD transforms and software modules, different mappings frequently occur. Two or even three bubbles can be combined and represented as one component, or a single bubble may be expanded to two or more components. Practical considerations and measures

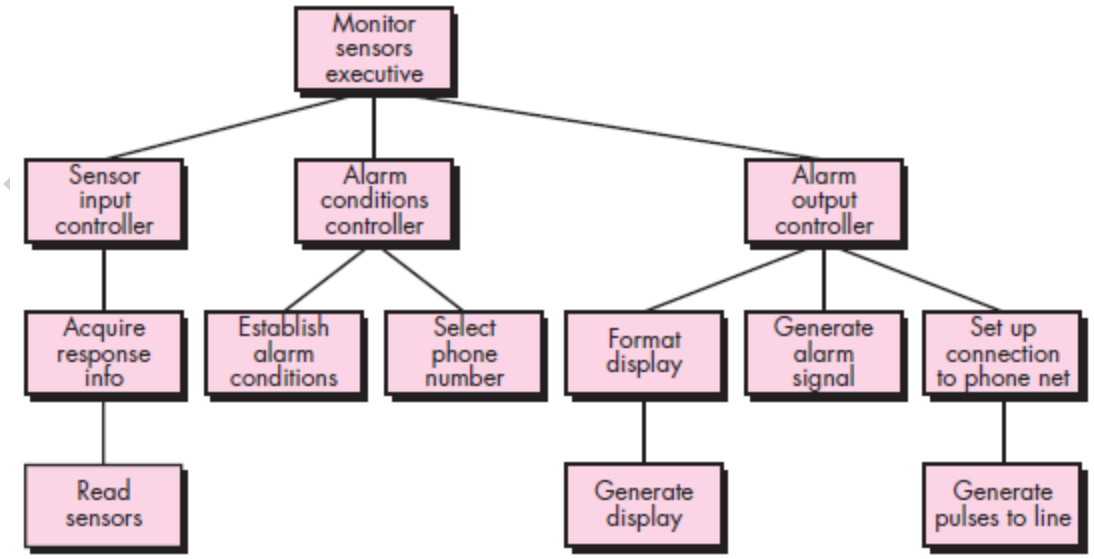


Second-level factoring for *monitor sensors*

of design quality dictate the outcome of second-level factoring. Review and refinement may lead to changes in this structure, but it can serve as a "first-iteration" design.

Second-level factoring for incoming flow follows in the same manner. Factoring is again accomplished by moving outward from the transform center boundary on the incoming flow side. The transform center of *monitor sensors* subsystem software is mapped somewhat differently. Each of the data conversion or calculation transforms of the transform portion of the DFD is mapped into a module subordinate to the transform controller. A completed first-iteration architecture is shown in Figure.

The components mapped in the preceding manner and shown in Figure 9.16 represent an initial design of software architecture. Although components are named in a manner that implies function, a brief processing narrative (adapted from the process specification developed for a data transformation created during requirements modeling) should be written for each. The narrative describes the



First-iteration structure for *monitor sensors*

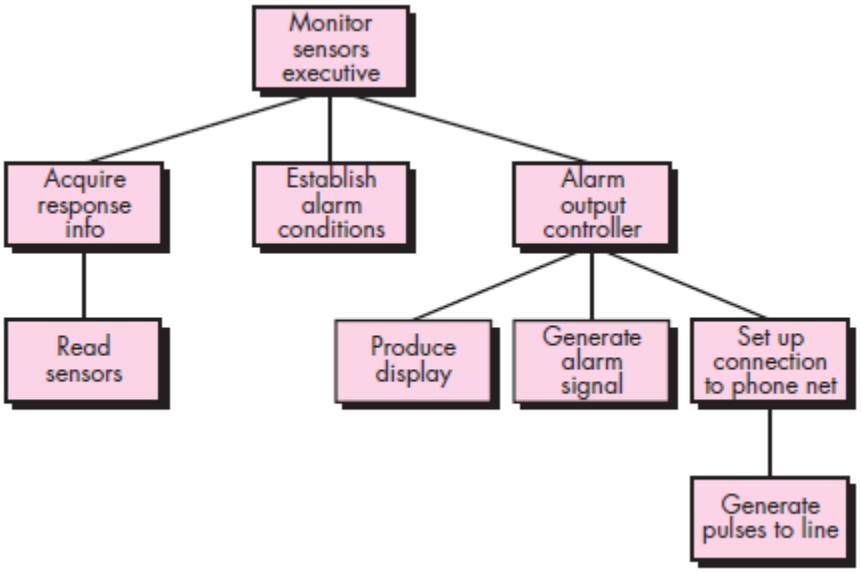
component interface, internal data structures, a functional narrative, and a brief discussion of restrictions and special features (e.g., file input-output, hardwaredependent characteristics, special timing requirements).

Step 7. Refine the first-iteration architecture using design heuristics for improved software quality. A first-iteration architecture can always be refined by applying concepts of functional independence (Chapter 8). Components are exploded or imploded to produce sensible factoring, separation of concerns, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.

Refinements are dictated by the analysis and assessment methods described, as well as practical considerations and common sense. There are times, for example, when the controller for incoming data flow is totally unnecessary, when some input processing is required in a component that is subordinate to the transform controller, when high coupling due to global data cannot be avoided, or when optimal structural characteristics cannot be achieved. Software requirements coupled with human judgment is the final arbiter.

The objective of the preceding seven steps is to develop an architectural representation of software. That is, once structure is defined, we can evaluate and refine software architecture by viewing it as a whole. Modifications made at this time require little additional work, yet can have a profound impact on software quality.

You should pause for a moment and consider the difference between the design approach described and the process of "writing programs." If code is the only representation of software, you and your colleagues will have great difficulty evaluating or refining at a global or holistic level and will, in fact, have difficulty "seeing the forest for the trees."



Refined program structure for *monitor sensors*

Refining the Architectural Design

Any discussion of design refinement should be prefaced with the following comment: "Remember that an 'optimal design' that doesn't work has questionable merit." You should be concerned with developing a representation of software that will meet all functional and performance requirements and merit acceptance based on design measures and heuristics.

Refinement of software architecture during early stages of design is to be encouraged. As I discussed earlier in this chapter, alternative architectural styles may be derived, refined, and evaluated for the "best" approach. This approach to optimization is one of the true benefits derived by developing a representation of software architecture.

It is important to note that structural simplicity often reflects both elegance and efficiency. Design refinement should strive for the smallest number of components that is consistent with effective modularity and the least complex data structure that adequately serves information requirements.

POSSIBLE QUESTIONS

PART B

1. Discuss about Design process with the Context of Software Engineering.
2. Discuss about quality guidelines and attributes in detail.
3. Explain the different levels of design pyramid in software architecture.
4. Explain in detail about Design Concepts.
5. Explain architectural design with appropriate pictorial representations.
6. Elucidate the principles to assessing alternative architectural design
7. Discuss about quality guidelines and attributes to evaluate the good design.
8. Illustrate the transform mapping steps to map data flow diagram into architecture.

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS: III BSC CS COURSE NAME: SOFTWARE ENGINEERING
COURSE CODE: 15CSU601 UNIT: III BATCH-2015-2018

ONE MARKS

	Questions	Opt1	Opt2	Opt3	Opt4	Answer
1	There are _____ major phases to any design process	2	3	4	5	2
2	Diversification is the _____ of a repertoire of alternatives.	component	solution	acquisition	knowledge	acquisition
3	During _____, the designer chooses and combines appropriate elements from the repertoire to meet the design objectives.	diversification	convergence	elimination	creation	convergence
4	_____ and _____ combine intuition and judgement based on experience in building similar entities.	elimination, convergence	creation, convergence	acquisition, creation	diversification and convergence	diversification and convergence
5	_____ can be traced to a customer's requirements and at the same time assessed for quality against a set of predefined criteria.	design	analysis	principles	testing	design
6	The _____ must implement all of the explicit requirements contained in the analysis model	principles	testing	design	component	design
7	A _____ should exhibit an architectural structure that has been created using recognizable design patterns.	principles	testing	component	design	design
8	A _____ is composed of components that exhibit good design characteristics.	principles	testing	component	design	design
9	A _____ can be implemented in an evolutionary fashion thereby facilitating implementation and testing.	principles	testing	component	design	design

10	A _____ should be modular that is the software should be logically partitioned into elements that perform specific functions and sub functions.	design	principles	component	testing	design
11	A _____ should contain distinct representations of data, architecture, interfaces, and components.	design	principles	component	testing	design
12	A _____ should lead to data structures that are appropriate for the objects to be implemented and are drawn from recognizable data patterns.	design	principles	component	testing	design
13	A _____ should lead to interfaces that reduce the complexity of connections between modules and with the external environment.	design	principles	component	testing	design
14	A _____ should be derived using a repeatable method that is driven by information obtained during software requirements analysis	principles	component	design	testing	design
15	The software _____ process encourages good design through the application of fundamental design principles, systematic methodology and thorough review.	principles	component	design	testing	design
16	The _____ must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.	principles	component	design	testing	design
17	The _____ should provide a complete picture of the software addressing the data, functional and behavioral domains from an implementation perspective.	principles	component	design	testing	design
18	The evolution of software _____ is a continuing process that has spanned the past four decades.	principles	component	design	testing	design

19	Procedural aspects of design definition evolved into a philosophy called _____.	top down programming	bottom up programming	structured programming	object oriented programming	structured programming
20	The design process should not suffer from _____.	analysis	tunnel vision	conceptual errors	integrity	tunnel vision
21	The design should be _____ to the analysis model.	consistent	related	traceable	relevant	traceable
22	The design should not _____ the wheel.	minimize	maximize	integrate	reinvent	reinvent
23	The design should _____ the intellectual distance	maximize	minimize	integrate	analyse	minimize
24	. The _____ is represented at a high level of abstraction	specification	analysis	quality	design specification	design specification
25	The design should exhibit _____ and integration.	uniformity	analysis	quality	review	uniformity
26	The design should be _____ to accommodate change.	reviewed	analysed	assessed	structured	structured
27	The design should be _____ to degrade gently, even when aberrant data, events, or operating conditions are encountered.	reviewed	analysed	assessed	structured	structured
28	Design is not _____, coding is not design	coding	analysis	review	event	coding
29	Design is not coding, _____ is not design.	coding	analysis	review	event	coding
30	The design should be _____ for quality as it is being created not after the fact.	reviewed	assessed	structured	integrated	assessed
31	The design should be _____ to minimize conceptual errors.	reviewed	assessed	structured	integrated	reviewed
32	Software design is both a _____ and a model.	model	process	data	function	process
33	_____ is the only way that we can accurately translate a customer's requirements into a finished software product or system.	specification	design	data	prototype	design
34	The design _____ is the equivalent of an architect's plan for a house.	analysis	process	model	function	model

35	At the highest level of _____, a solution is stated in broad terms, using the language of the problem environment.	refinement	modularity	abstraction	continuity	abstraction
36	A _____ is a named sequence of instructions that has a specific and limited function.	procedural abstraction	data abstraction	control abstraction	Process abstraction	procedural abstraction
37	A _____ is a named collection of data that describes a data object.	procedural abstraction	data abstraction	control abstraction	Process abstraction	data abstraction
38	_____ implies a program control mechanism without specifying internal detail.	procedural abstraction	data abstraction	control abstraction	Process abstraction	control abstraction
39	_____ is used to coordinate activities in an operating system.	synchronization semaphore	control abstraction	data abstraction	procedural abstraction	synchronization semaphore
40	_____ is a top down design strategy originally proposed by Niklaus Wirth.	stepwise refinement	control abstraction	data abstraction	procedural abstraction	stepwise refinement
41	The designer's goal is to produce a model or representation of a _____ that will later be built	component	entity	data	raw material	component
42	The second phase of any design process is the gradual _____ of all but one particular configuration of components, and thus the creation of the final product.	acquisition	addition	elimination	creation	elimination
43	Design begins with the _____ model.	data	requirements	specification	code	requirements
44	Software design methodologies lack the _____ that are normally associated with more classical engineering design disciplines.	depth	flexibility	quantitative nature	all of the above	all of the above
45	Software requirements, manifested by the _____ models, feed the design task.	data	functional	behavioral	all of the above	all of the above
46	_____ is the place where quality is fostered in software engineering	model	data	design	specification	design
47	_____ provides us with representations of software that can be assessed for quality.	design	specification	data	prototype	design
48	Procedural aspects of design definition evolved into a philosophy called _____.	procedural programming	object oriented programming	structured programming	all of the above	structured programming

49	Meyer defines _____ criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system.	2	3	4	5	5
50	If a design method provides a systematic mechanism for decomposing the problem into sub problems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution. This is called _____.	modular decomposability	modular composability	modular understandability	modular continuity	modular decomposability
51	If a design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel. This is called _____.	modular decomposability	modular composability	modular understandability	modular continuity	modular composability
52	If a module can be understood as a stand alone unit (without reference to other modules), it will be easier to build and easier to change. This is called _____.	modular decomposability	modular composability	modular understandability	modular continuity	modular understandability
53	If small changes to the system requirements result in changes to individual modules, rather than system wide changes, the impact of change-induced side effects will be minimized. This is called _____.	modular decomposability	modular composability	modular understandability	modular continuity	modular continuity
54	If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized. This is called _____.	modular protection	modular composability	modular understandability	modular continuity	modular protection
55	The aspect of the architectural design representation defines the components of a system and the manner in which those components are packaged and interact with one another. This property is called _____.	extra functional property	structural property	families of related systems	none of the above	structural property

56	_____ represent architecture as an organized collection of program components.	dynamic models	functional models	framework models	structural models	structural models
57	_____ increases the level of design abstraction by attempting to identity repeatable architectural design frameworks that are encountered in similar types of applications.	framework models	dynamic models	process models	functional models	framework models
58	_____ address the behavioural aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.	framework models	dynamic models	process models	functional models	dynamic models
59	_____ focus on the design of the business or technical process that the system must accommodate.	framework models	dynamic models	process models	functional models	process models
60	_____ can be used to represent the functional hierarchy of a system.	framework models	dynamic models	process models	functional models	functional models

UNIT-IV

SYLLABUS

Performing User Interface Design: The Golden Rules: Place the User in Control-Reduce the User's Memory Load-Make the Interface Consistent- User Interface Analysis and Design: Interface Analysis and Design Models- The Process- Interface Analysis: User Analysis - Task analysis and Modeling. Interface Design Concepts-Applying Interface Design Steps-User Interface Design Patterns-Design Issues –Design Evolution.

PERFORMING USER INTERFACE DESIGN

We live in a world of high-technology products, and virtually all of them—consumer electronics, industrial equipment, corporate systems, military systems, personal computer software, and WebApps—require human interaction. If a product is to be successful, it must exhibit good *usability*— a qualitative measure of the ease and efficiency with which a human can employ the functions and features offered by the high-technology product.

Whether an interface has been designed for a digital music player or the weapons control system for a fighter aircraft, usability matters. If interface mechanisms have been well designed, the user glides through the interaction using a smooth rhythm that allows work to be accomplished effortlessly. But if the interface is poorly conceived, the user moves in fits and starts, and the end result is frustration and poor work efficiency.

For the first three decades of the computing era, usability was not a dominant concern among those who built software. In his classic book on design, Donald Norman argued that it was time for a change in attitude:

To make technology that fits human beings, it is necessary to study human beings. But now we tend to study only the technology. As a result, people are required to conform to technology. It is time to reverse this trend, time to make technology that conforms to people.

As technologists studied human interaction, two dominant issues arose. First, a set of *golden rules* were identified. These applied to all human interaction with technology products. Second, a set of *interaction mechanisms* were defined to enable software designers to build systems that properly implemented the golden rules. These interaction mechanisms, collectively called the *graphical user interface* (GUI), have eliminated some of the most egregious problems associated with human interfaces.

But even in a "Windows world," we all have encountered user interfaces that are difficult to learn, difficult to use, confusing, counterintuitive, unforgiving, and in many cases, totally frustrating. Yet, someone spent time and energy building each of these interfaces, and it is not likely that the builder created these problems purposely.

THE GOLDEN RULES

In his book on interface design, Theo Mandel [Man97] coins three *golden rules:*

1. Place the user in control.

2. Reduce the user's memory load.

3. Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important aspect of software design.

PLACE THE USER IN CONTROL

During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the window-oriented graphical interface. "What I really would like," said the user solemnly, "is a system that reads my mind. It knows what I want to do before I need to do it and makes it very easy for me to get it done. That's all, just that." My first reaction was to shake my head and smile, but I paused for a moment.

There was absolutely nothing wrong with the user's request. She wanted a system that reacted to her needs and helped her get things done. She wanted to control the computer, not have the computer control her.

Most interface constraints and restrictions that are imposed by a designer are intended to simplify the mode of interaction. But for whom?

As a designer, you may be tempted to introduce constraints and limitations to simplify the implementation of the interface. The result may be an interface that is easy to build, but frustrating to use. Mandel defines a number of design principles that allow the user to maintain control:

Define interaction modes in a way that does not force a user into unnecessary or undesired actions.

An interaction mode is the current state of the interface. For example, if *spell check* is selected in a word-processor menu, the software moves to a spell-checking mode. There is no reason to force the user to remain in spell-checking mode if the user desires to make a small text edit along the way. The user should be able to enter and exit the mode with little or no effort.

Provide for flexible interaction.

Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multitouch screen, or voice recognition commands. But every action is not amenable to every interaction mechanism. Consider, for example, the difficulty of using keyboard command (or voice input) to draw a complex shape.

Allow user interaction to be interruptible and undoable.

Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action.

Streamline interaction as skill levels advance and allow the interaction to be customized.

Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.

Hide technical internals from the casual user.

The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology. In essence, the interface should never require that the user interact at a level that is "inside" the machine (e.g., a user should never be required to type operating system commands from within application software).

Design for direct interaction with objects that appear on the screen.

The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to "stretch" an object (scale it in size) is an implementation of direct manipulation.

REDUCE THE USER'S MEMORY LOAD

The more a user has to remember, the more error-prone the interaction with the system will be. It is for this reason that a well-designed user interface does not tax the user's memory. Whenever possible, the system should "remember" pertinent information and assist the user with an interaction scenario that assists recall. Mandel [Man97] defines design principles that enable an interface to reduce the user's memory load:

Reduce demand on short-term memory.

When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results.

This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.

Establish meaningful defaults.

The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a "reset" option should be available, enabling the redefinition of original default values.

Define shortcuts that are intuitive.

When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to be invoked).

The visual layout of the interface should be based on a real-world metaphor.

For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

Disclose information in a progressive fashion.

The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick. An example, common to many word-processing applications, is the underlining function.

The function itself is one of a number of functions under a *text style* menu. However, every underlining capability is not listed. The user must pick underlining; then all underlining options (e.g., single underline, double underline, dashed underline) are presented.

MAKE THE INTERFACE CONSISTENT

The interface should present and acquire information in a consistent fashion. This implies that (1) all visual information is organized according to design rules that are maintained throughout all screen displays, (2) input mechanisms are constrained to a limited set that is used consistently throughout the application, and (3) mechanisms for navigating from task to task are

consistently defined and implemented. Mandel [Man97] defines a set of design principles that help make the interface consistent:

Allow the user to put the current task into a meaningful context.

Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

Maintain consistency across a family of applications.

A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction.

If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application he encounters. A change (e.g., using alt-S to invoke scaling) will cause confusion. The interface design principles discussed in this and the preceding sections provide you with basic guidance. In the sections that follow, you'll learn about the interface design process itself.

USER INTERFACE ANALYSIS AND DESIGN

The overall process for analyzing and designing a user interface begins with the creation of different models of system function (as perceived from the outside). You begin by delineating the human- and computer-oriented tasks that are required to achieve system function and then considering the design issues that apply to all interface designs. Tools are used to prototype and ultimately implement the design model, and the result is evaluated by end users for quality.

INTERFACE ANALYSIS AND DESIGN MODELS

Four different models come into play when a user interface is to be analyzed and designed. A human engineer (or the software engineer) establishes a *user model*, the software engineer creates a *design model*, the end user develops a mental image that is often called the user's *mental model* or the *system perception*, and the implementers

of the system create an *implementation model*. Unfortunately, each of these models may differ significantly. Your role, as an interface designer, is to reconcile these differences and derive a consistent representation of the interface.

The user model establishes the profile of end users of the system. In his introductory column on "user-centric design," Jeff Patton [Pat07] notes:

The truth is, designers and developers—myself included—often think about users. However, in the absence of a strong mental model of specific users, we self-substitute. Selfsubstitution isn't user centric—it's self-centric.

To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic background, motivation, goals and personality" [Shn04]. In addition, users can be categorized as:

Novices. No syntactic knowledge1 of the system and little semantic knowledge2 of the application or computer usage in general.

Knowledgeable, intermittent users. Reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface.

Knowledgeable, frequent users. Good semantic and syntactic knowledge that often leads to the "power-user syndrome"; that is, individuals who look for shortcuts and abbreviated modes of interaction.

The user's *mental model* (system perception) is the image of the system that end users carry in their heads. For example, if the user of a particular word processor were asked to describe its operation, the system perception would guide the response. The accuracy of the description will depend upon the user's profile (e.g., novices would provide a sketchy response at best) and overall familiarity with software in the application domain. A user who understands word processors fully but has worked with the specific word processor only once might actually be able to provide a more complete description of its function than the novice who has spent weeks trying to learn the system.

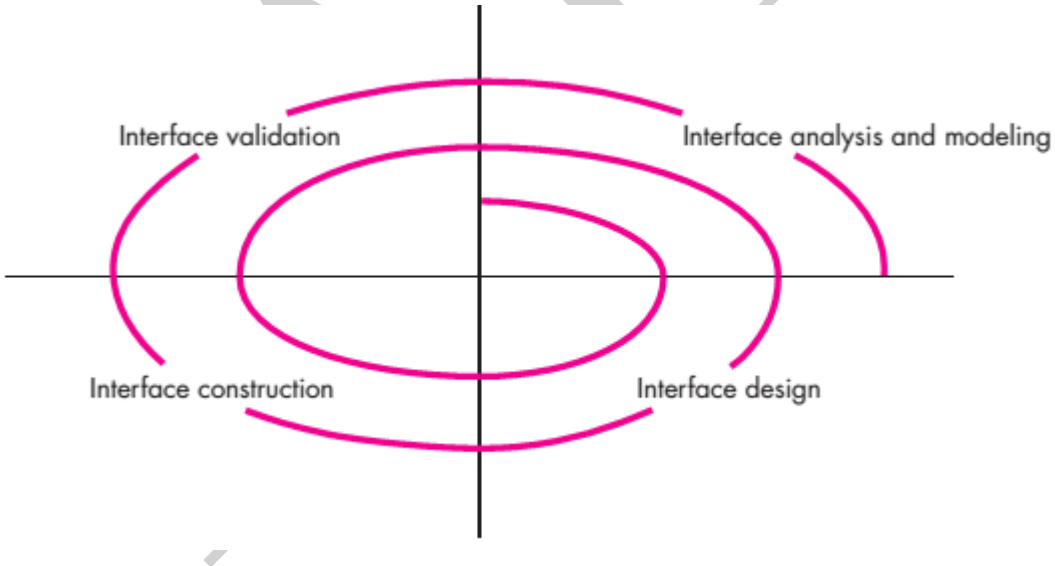
The *implementation model* combines the outward manifestation of the computerbased system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes, help files) that describes interface syntax and semantics. When the implementation model and the user's mental model are coincident, users generally feel comfortable with the software and use it effectively. To accomplish this "melding" of the models, the design model must have been

THE PROCESS

The analysis and design process for user interfaces is iterative and can be represented using a spiral model similar to the one discussed in Chapter 2. Referring to Figure 11.1, the user interface analysis and design process begins at the interior of the spiral and encompasses four distinct framework activities [Man97]: (1) interface analysis and modeling, (2) interface design, (3) interface construction, and (4) interface validation. The spiral shown in Figure 11.1 implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design. In most cases, the construction activity involves prototyping—the only practical way to validate what has been designed.

Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, you work to understand the system perception (Section 11.2.1) for each class of users.

Once general requirements have been defined, a more detailed *task analysis* is conducted. Those tasks that the user performs to accomplish the goals of the system.



The user interface design process

are identified, described, and elaborated (over a number of iterative passes through the spiral). Task analysis is discussed in more detail in Section 11.3. Finally, analysis of the user environment focuses on the physical work environment. Among the questions to be asked are

• Where will the interface be located physically?

• Will the user be sitting, standing, or performing other tasks unrelated to the interface?

• Does the interface hardware accommodate space, light, or noise constraints?

• Are there special human factors considerations driven by environmental factors?

The information gathered as part of the analysis action is used to create an analysis model for the interface. Using this model as a basis, the design action commences. The goal of *interface design* is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

Interface construction normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.

Interface validation focuses on (1) the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements; (2) the degree to which the interface is easy to use and easy to learn, and (3) the users' acceptance of the interface as a useful tool in their work.

As I have already noted, the activities described in this section occur iteratively. Therefore, there is no need to attempt to specify every detail (for the analysis or design model) on the first pass. Subsequent passes through the process elaborate task detail, design information, and the operational features of the interface.

INTERFACE ANALYSIS

A key tenet of all software engineering process models is this: *understand the problem before you attempt to design a solution.* In the case of user interface design, understanding the problem means understanding (1) the people (end users) who will interact with the system through the interface, (2) the tasks that end users must

USER ANALYSIS

The phrase "user interface" is probably all the justification needed to spend some time understanding the user before worrying about technical matters. Earlier I noted that each user has a mental image of the software that may be different from the mental image developed by other users. In addition, the user's mental image may be vastly different from the software engineer's design model. The only way that you can get the mental image and the design model to converge is to work to understand the users themselves as well as how these people will use the system. Information from a broad array of sources can be used to accomplish this:

User Interviews. The most direct approach, members of the software team meet with end users to better understand their needs, motivations, work culture, and a myriad of other issues. This can be accomplished in one-on-one meetings or through focus groups.

Sales input. Sales people meet with users on a regular basis and can gather information that will help the software team to categorize users and better understand their requirements.

Marketing input. Market analysis can be invaluable in the definition of market segments and an understanding of how each segment might use the software in subtly different ways.

Support input. Support staff talks with users on a daily basis. They are the most likely source of information on what works and what doesn't, what users like and what they dislike, what features generate questions and what features are easy to use.

The following set of questions (adapted from [Hac98]) will help you to better understand the users of a system:

• Are users trained professionals, technicians, clerical, or manufacturing workers?

• What level of formal education does the average user have?

• Are the users capable of learning from written materials or have they expressed a desire for classroom training?

• Are users expert typists or keyboard phobic?

• What is the age range of the user community?

• Will the users be represented predominately by one gender?

• How are users compensated for the work they perform?

• Do users work normal office hours or do they work until the job is done?

• Is the software to be an integral part of the work users do or will it be used only occasionally?

• What is the primary spoken language among users?

• What are the consequences if a user makes a mistake using the system?

• Are users experts in the subject matter that is addressed by the system?

• Do users want to know about the technology that sits behind the interface? Once these questions are answered, you'll know who the end users are, what is likely to motivate and please them, how they can be grouped into different user classes or profiles, what their mental models of the system are, and how the user interface must be characterized to meet their needs.

TASK ANALYSIS AND MODELING

The goal of task analysis is to answer the following questions:

• What work will the user perform in specific circumstances?

• What tasks and subtasks will be performed as the user does the work?

• What specific problem domain objects will the user manipulate as work is performed?

• What is the sequence of work tasks—the workflow?

• What is the hierarchy of tasks?

To answer these questions, you must draw upon techniques that I have discussed earlier in this book, but in this instance, these techniques are applied to the user interface.

Use cases.

The use case describes the manner in which an actor (in the context of user interface design, an actor is always a person) interacts with a system. When used as part of task analysis, the use case is developed to show how an end user performs some specific work-related task. In most instances, the use case is written in an informal style (a simple paragraph) in the first-person. For example, assume that a small software company wants to build a computer-aided design system explicitly for interior designers. To get a better understanding of how they do their work, actual interior designers are asked to describe a specific design function. When asked:

"How do you decide where to put furniture in a room?" an interior designer writes the following informal use case:

Begin by sketching the floor plan of the room, the dimensions and the location of windows and doors. I'm very concerned about light as it enters the room, about the view out of the windows (if it's beautiful, I want to draw attention to it), about the running length of an unobstructed wall, about the flow of movement through the room. I then look at the list of furniture my customer and I have chosen—tables, chairs, sofa, cabinets, the list of accents—lamps, rugs, paintings, sculpture, plants, smaller pieces, and my notes on any desires my customer has for placement. I then draw each item from my lists using a template that is scaled to the floor plan. I label each item I draw and use pencil because I always move things.

Consider a number of alternative placements and decide on the one I like best. Draw a rendering (a 3-D picture) of the room to give my customer a feel for what it'll look like.

This use case provides a basic description of one important work task for the computer-aided design system. From it, you can extract tasks, objects, and the overall flow of the interaction. In addition, other features of the system that would please the interior designer might also be conceived. For example, a digital photo could be taken looking out each window in a room. When the room is rendered, the actual outside view could be represented through each window.

Task elaboration.

Here stepwise elaboration is done (also called functional decomposition or stepwise refinement) as a mechanism for refining the processing tasks that are required for software to accomplish some desired function.

Task analysis for interface design uses an elaborative approach to assist in understanding the human activities the user interface must accommodate.

Task analysis can be applied in two ways. An interactive, computer-based system is often used to replace a manual or semimanual activity. To understand the tasks that must be performed to accomplish the goal of the activity, you must understand the tasks that people currently perform (when using a manual approach) and then map these into a similar (but not necessarily identical) set of tasks that are implemented in the context of the user interface. Alternatively, you can study an existing specification for a computer-based solution and derive a set of user tasks that will accommodate the user model, the design model, and the system perception.

Regardless of the overall approach to task analysis, you must first define and classify tasks. I have already noted that one approach is stepwise elaboration. For example, let's reconsider the computer-aided design system for interior designers discussed earlier. By observing an interior designer at work, you notice that interior design comprises a number of major activities: furniture layout (note the use case discussed earlier), fabric and material selection, wall and window coverings selection, presentation (to the customer), costing, and shopping. Each of these major tasks can be elaborated into subtasks.

For example, using information contained in the use case, furniture layout can be refined into the following tasks:

(1) draw a floor plan based on room dimensions,

(2) place windows and doors at appropriate locations,

(3a) use furniture templates to draw scaled furniture outlines on the floor plan,

(3b) use accents templates to draw scaled accents on the floor plan,

(4) move furniture outlines and accent outlines to get the best placement,

(5) label all furniture and accent outlines,

(6) draw dimensions to show location, and

(7) draw a perspective-rendering view for the customer.

A similar approach could be used for each of the other major tasks. Subtasks 1 to 7 can each be refined further. Subtasks 1 to 6 will be performed by manipulating information and performing actions within the user interface. On the other hand, subtask 7 can be performed automatically in software and will result in little direct user interaction.4 The design model of the interface should accommodate each of these tasks in a way that is consistent with the user model (the profile of a "typical" interior designer) and system perception (what the interior designer expects from an automated system).

Object elaboration.

Rather than focusing on the tasks that a user must perform, you can examine the use case and other information obtained from the user and extract the physical objects that are used by the interior designer. These objects can

be categorized into classes. Attributes of each class are defined, and an evaluation of the actions applied to each object provide a list of operations. For example, the furniture template might translate into a class called **Furniture** with attributes that might include size, shape,

location, and others. The interior designer would *select* the object from the **Furniture** class, *move* it to a position on the floor plan (another object in this context), *draw* the furniture outline, and so forth. The tasks *select*, *move*, and *draw* are operations. The user interface analysis model would not provide a literal implementation for each of these operations. However, as the design is elaborated, the details of each operation are defined.

Workflow analysis.

When a number of different users, each playing different roles, makes use of a user interface, it is sometimes necessary to go beyond task analysis and object elaboration and apply *workflow analysis.* This technique allows you to understand how a work process is completed when several people (and roles) are involved. Consider a company that intends to fully automate the process of prescribing and delivering prescription drugs. The entire process5 will revolve around a

Web-based application that is accessible by physicians (or their assistants), pharmacists, and patients. Workflow can be represented effectively with a UML swimlane diagram (a variation on the activity diagram).

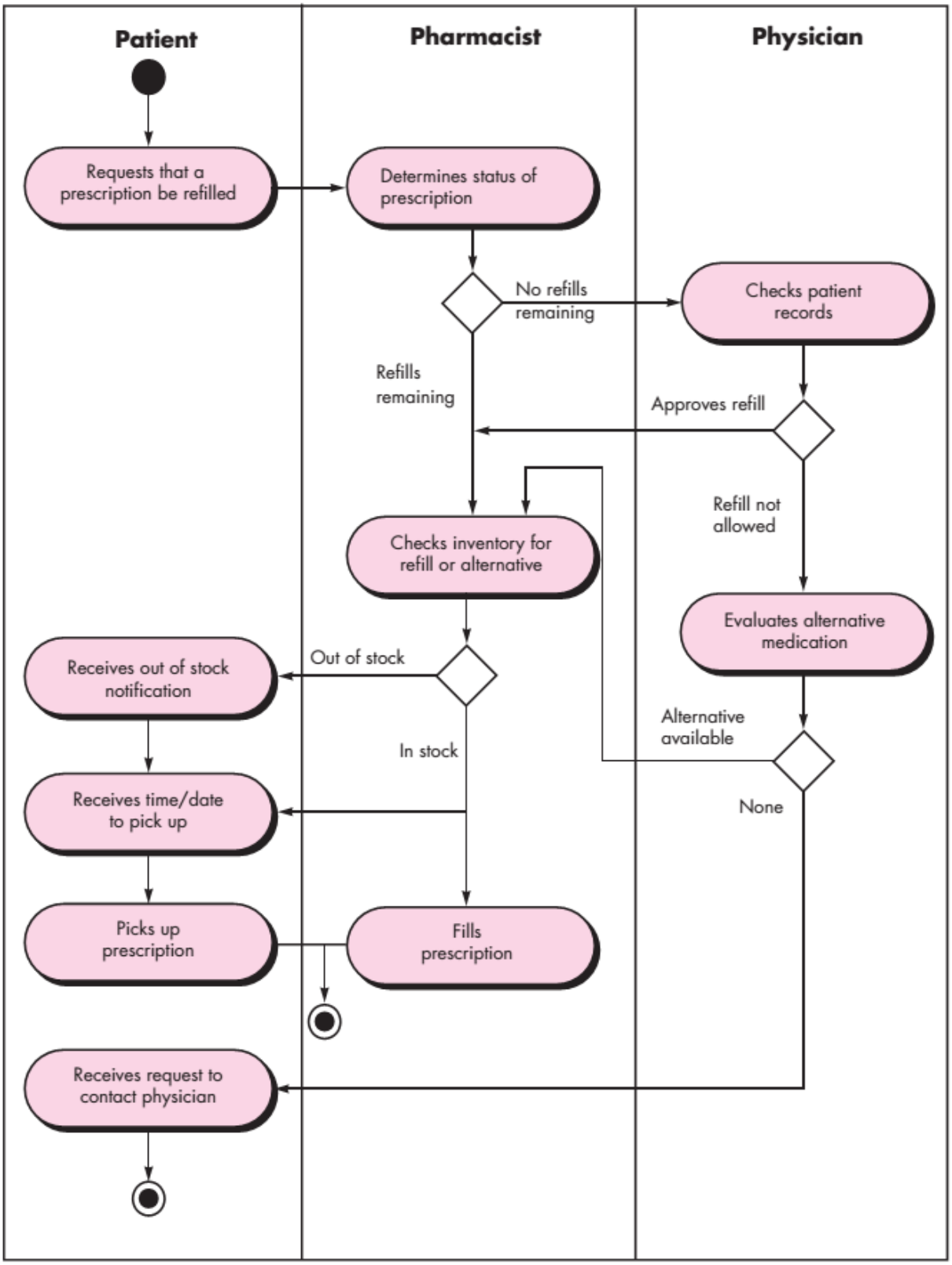
We consider only a small part of the work process: the situation that occurs when a patient asks for a refill. Figure 11.2 presents a swimlane diagram that indicates the tasks and decisions for each of the three roles noted earlier. This information may have been elicited via interview or from use cases written by each actor. Regardless, the flow of events (shown in the figure) enables you to recognize a number of key interface characteristics:

1. Each user implements different tasks via the interface; therefore, the look and feel of the interface designed for the patient will be different than the one defined for pharmacists or physicians.

2. The interface design for pharmacists and physicians must accommodate access to and display of information from secondary information sources (e.g., access to inventory for the pharmacist and access to information about alternative medications for the physician).

3. Many of the activities noted in the swimlane diagram can be further elaborated using task analysis and/or object elaboration (e.g., *Fills prescription* could imply a mail-order delivery, a visit to a pharmacy, or a visit to a special drug distribution center).

Hierarchical representation. A process of elaboration occurs as you begin to analyze the interface. Once workflow has been established, a task hierarchy can be defined for each user type. The hierarchy is derived by a stepwise elaboration of


Patient
Pharmacist
Physician
Requests that a prescription be refilled
Determines status of prescription
No refills remaining
Checks patient records
Refills remaining
Approves refill
Refill not allowed
Checks inventory for refill or alternative
Evaluates alternative medication
Out of stock
Receives out of stock notification
In stock
Alternative available
None
Receives time/date to pick up
Picks up prescription
Fills prescription
Receives request to contact physician

Swimlane diagram for prescription refill function each task identified for the user. For example, consider the following user task and subtask hierarchy.

User task: ***Requests that a prescription be refilled***

• *Provide identifying information.*

• *Specify name.*

• *Specify userid.*

• *Specify PIN and password.*

• *Specify prescription number.*

• *Specify date refill is required.*

To complete the task, three subtasks are defined. One of these subtasks, *provide identifying information*, is further elaborated in three additional sub-subtasks.

Analysis of Display Content

The user tasks identified lead to the presentation of a variety of different types of content. For modern applications, display content can range from character-based reports (e.g., a spreadsheet), graphical displays (e.g., a histogram, a 3-D model, a picture of a person), or specialized information (e.g., audio or video files). The analysis modeling techniques identify the output data objects that are produced by an application. These data objects may be

(1) generated by components (unrelated to the interface) in other parts of an application,

(2) acquired from data stored in a database that is accessible from the application, or

(3) transmitted from systems external to the application in question.

During this interface analysis step, the format and aesthetics of the content (as it is displayed by the interface) are considered. Among the questions that are asked and answered are:

• Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right-hand corner)?

• Can the user customize the screen location for content?

• Is proper on-screen identification assigned to all content?

• If a large report is to be presented, how should it be partitioned for ease of understanding?

• Will mechanisms be available for moving directly to summary information for large collections of data?

• Will graphical output be scaled to fit within the bounds of the display device that is used?

• How will color be used to enhance understanding?

• How will error messages and warnings be presented to the user?

The answers to these (and other) questions will help you to establish requirements for content presentation.

Analysis of the Work Environment

Hackos and Redish [Hac98] discuss the importance of work environment analysis when they state:

People do not perform their work in isolation. They are influenced by the activity around them, the physical characteristics of the workplace, the type of equipment they are using, and the work relationships they have with other people. If the products you design do not fit into the environment, they may be difficult or frustrating to use. In some applications the user interface for a computer-based system is placed in a "user-friendly location" (e.g., proper lighting, good display height, easy keyboard access), but in others (e.g., a factory floor or an airplane cockpit), lighting may be suboptimal, noise may be a factor, a keyboard or mouse may not be an option, display placement may be less than ideal. The interface designer may be constrained by factors that mitigate against ease of use.

In addition to physical environmental factors, the workplace culture also comes into play. Will system interaction be measured in some manner (e.g., time per transaction or accuracy of a transaction)? Will two or more people have to share information before an input can be provided? How will support be provided to users of the system? These and many related questions should be answered before the interface design commences.

INTERFACE DESIGN CONCEPTS:

APPLYING INTERFACE DESIGN STEPS

Once interface analysis has been completed, all tasks (or objects and actions) required by the end user have been identified in detail and the interface design activity commences. Interface design, like all software engineering design, is an iterative process. Each user interface design step occurs a number of times, elaborating and refining information developed in the preceding step.

Although many different user interface design models (e.g., [Nor86], [Nie00]) have been proposed, all suggest some combination of the following steps:

1. Using information developed during interface analysis (Section 11.3), define interface objects and actions (operations).

2. Define events (user actions) that will cause the state of the user interface to change. Model this behavior.

3. Depict each interface state as it will actually look to the end user.

4. Indicate how the user interprets the state of the system from information provided through the interface.

In some cases, you can begin with sketches of each interface state (i.e., what the user interface looks like under various circumstances) and then work backward to define objects, actions, and other important design information. Regardless of the sequence of design tasks, you should (1) always follow the golden rules discussed in Section 11.1, (2) model how the interface will be implemented, and (3) consider the environment (e.g., display technology, operating system, development tools) that will be used.

APPLYING INTERFACE DESIGN STEPS

The definition of interface objects and the actions that are applied to them is an important step in interface design. To accomplish this, user scenarios are parsed in much the same way as described in Chapter 6. That is, a use case is written. Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.

Once the objects and actions have been defined and elaborated iteratively, they are categorized by type. Target, source, and application objects are identified. A *source object* (e.g., a report icon) is dragged and dropped onto a *target object* (e.g., a printer icon). The implication

of this action is to create a hard-copy report. An *application object* represents application-specific data that are not directly manipulated as part of screen interaction. For example, a mailing list is used to store names for a mailing. The list itself might be sorted, merged, or purged (menu-based actions), but it is not dragged and dropped via user interaction.

When you are satisfied that all important objects and actions have been defined (for one design iteration), screen layout is performed. Like other interface design activities, screen layout is an interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and definition of major and minor menu items are conducted. If a real-world metaphor is appropriate for the application, it is specified at this time, and the layout is organized in a manner that complements the metaphor.

To provide a brief illustration of the design steps noted previously, consider a user scenario for the *SafeHome* system (discussed in earlier chapters). A preliminary use case (written by the homeowner) for the interface follows:

Preliminary use case:

I want to gain access to my *SafeHome* system from any remote location via the Internet. Using browser software operating on my notebook computer (while I'm at work or traveling), I can determine the status of the alarm system, arm or disarm the system, reconfigure security zones, and view different rooms within the house via preinstalled video cameras.

To access *SafeHome* from a remote location, I provide an identifier and a password. These define levels of access (e.g., all users may not be able to reconfigure the system) and provide security. Once validated, I can check the status of the system and change the status by arming or disarming *SafeHome.* I can reconfigure the system by displaying a floor plan of the house, viewing each of the security sensors, displaying each currently configured zone, and modifying zones as required. I can view the interior of the house via strategically placed video cameras. I can pan and zoom each camera to provide different views of the interior.

Based on this use case, the following homeowner tasks, objects, and data items are identified: • *accesses* the *SafeHome* system

• *enters* an **ID** and **password** to allow remote access

• *checks* **system status**

• *arms* or *disarms SafeHome* system

• *displays* **floor plan** and **sensor locations**

• *displays* **zones** on floor plan

- *changes* **zones** on floor plan
- *displays* **video camera locations** on floor plan
- *selects* **video camera** for viewing
- *views* **video images** (four frames per second)
- *pans* or *zooms* the **video camera**

Objects (boldface) and actions (italics) are extracted from this list of homeowner tasks. The majority of objects noted are application objects. However, **video camera location** (a source object) is dragged and dropped onto **video camera** (a target object) to create a **video image** (a window with video display).

A preliminary sketch of the screen layout for video monitoring is created (Figure 11.3).6 To invoke the video image, a video camera location icon, *C*, located in the floor plan displayed in the monitoring window is selected. In this case a camera location in the living room (LR) is then dragged and dropped onto the video camera icon in the upper left-hand portion of the screen. The video image window appears, displaying streaming video from the camera located in the LR. The zoom and pan control slides are used to control the magnification and direction of the video image.

To select a view from another camera, the user simply drags and drops a different camera location icon into the camera icon in the upper left-hand corner of the screen.

The layout sketch shown would have to be supplemented with an expansion of each menu item within the menu bar, indicating what actions are available for the video monitoring mode (state). A complete set of sketches for each homeowner task noted in the user scenario would be created during the interface design.

USER INTERFACE DESIGN PATTERNS

Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged. As I noted earlier in this book, a design pattern is

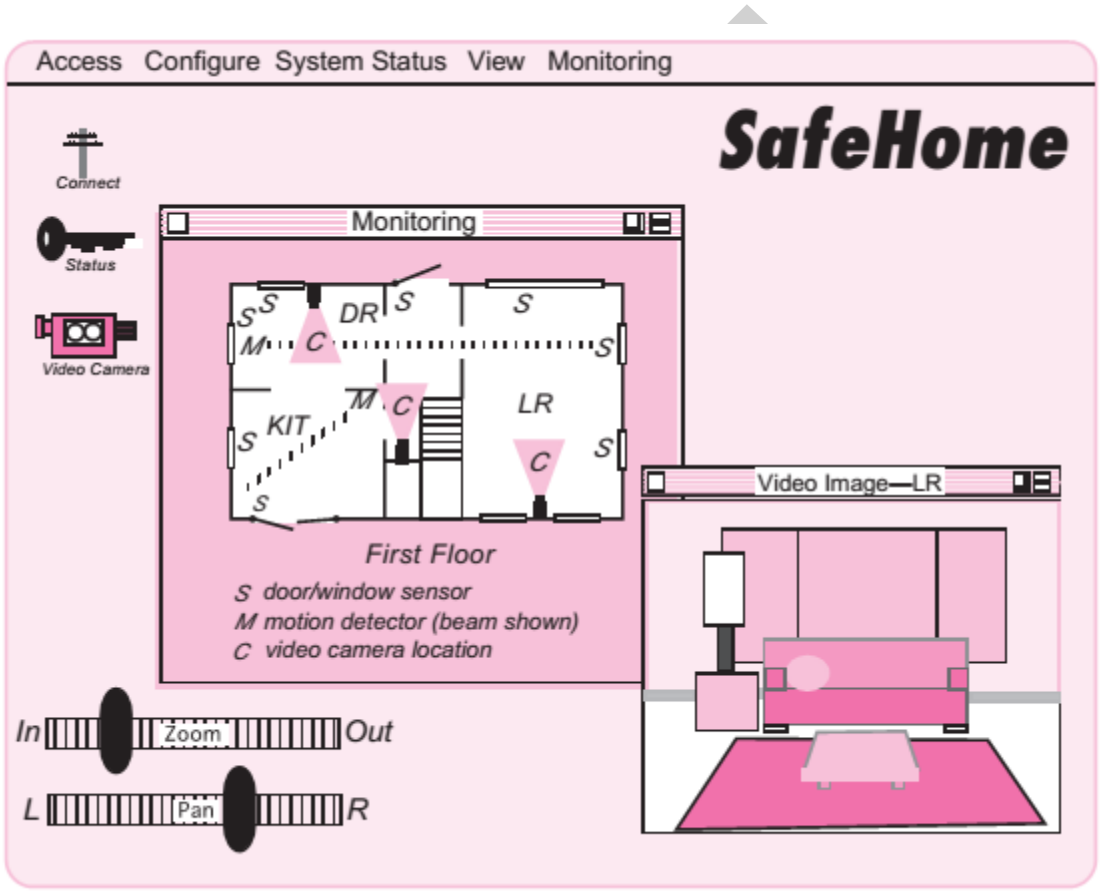


Fig. Preliminary screen layout

an abstraction that prescribes a design solution to a specific, well-bounded design problem.

As an example of a commonly encountered interface design problem, consider a situation in which a user must enter one or more calendar dates, sometimes months in advance. There are many possible solutions to this simple problem, and a number of different patterns that might be proposed. Laakso [Laa00] suggests a pattern called **CalendarStrip** that produces a continuous, scrollable calendar in which the current date is highlighted and future dates may be

selected by picking them from the calendar. The calendar metaphor is well known to every user and provides an effective mechanism for placing a future date in context.

A vast array of interface design patterns has been proposed over the past decade. A more detailed discussion of user interface design patterns is presented in Chapter 12. In addition, Erickson [Eri08] provides pointers to many Web-based collections.

DESIGN ISSUES

As the design of a user interface evolves, four common design issues almost always surface: system response time, user help facilities, error information handling, and command labeling. Unfortunately, many designers do not address these issues until relatively late in the design process (sometimes the first inkling of a problem doesn't occur until an operational prototype is available). Unnecessary iteration, project delays, and end-user frustration often result. It is far better to establish each as a design issue to be considered at the beginning of software design, when changes are easy and costs are low.

Response time. System response time is the primary complaint for many interactive applications. In general, system response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action. System response time has two important characteristics: length and variability. If system response is too long, user frustration and stress are inevitable.

Variability refers to the deviation from average response time, and in many ways, it is the most important response time characteristic. Low variability enables the user to establish an interaction rhythm, even if response time is relatively long. For example, a 1-second response to a command will often be preferable to a response that varies from 0.1 to 2.5 seconds. When variability is significant, the user is always off balance, always wondering whether something "different" has occurred behind the scenes.

Help facilities. Almost every user of an interactive, computer-based system requires help now and then. In some cases, a simple question addressed to a knowledgeable colleague can do the trick. In others, detailed research in a multivolume set of "user manuals" may be the only option. In most cases, however, modern software provides online help facilities that enable a user to get a question answered or resolve a problem without leaving the interface.

A number of design issues [Rub88] must be addressed when a help facility is considered:

• Will help be available for all system functions and at all times during system interaction? Options include help for only a subset of all functions and actions or help for all functions.

• How will the user request help? Options include a help menu, a special function key, or a HELP command.

• How will help be represented? Options include a separate window, a reference to a printed document (less than ideal), or a one- or two-line suggestion produced in a fixed screen location.

• How will the user return to normal interaction? Options include a return button displayed on the screen, a function key, or control sequence.

How will help information be structured? Options include a "flat" structure in which all information is accessed through a keyword, a layered hierarchy of information that provides increasing detail as the user proceeds into the structure, or the use of hypertext.

Error handling.

Error messages and warnings are "bad news" delivered to users of interactive systems when something has gone awry. At their worst, error messages and warnings impart useless or misleading information and serve only to increase user frustration. There are few computer users who have not encountered an error of the form: *"Application XXX has been forced to quit because an error of type 1023 has been encountered."* Somewhere, an explanation for error 1023 must exist; otherwise, why would the designers have added the identification? Yet, the error message provides no real indication of what went wrong or where to look to get additional information. An error message presented in this manner does nothing to assuage user anxiety or to help correct the problem.

In general, every error message or warning produced by an interactive system should have the following characteristics:

• The message should describe the problem in jargon that the user can understand.

• The message should provide constructive advice for recovering from the error.

• The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have).

• The message should be accompanied by an audible or visual cue. That is, a beep might be generated to accompany the display of the message, or the message might flash momentarily or be displayed in a color that is easily recognizable as the "error color."

• The message should be "nonjudgmental." That is, the wording should never place blame on the user.

Because no one really likes bad news, few users will like an error message no matter how well designed. But an effective error message philosophy can do much to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur.

Menu and command labeling.

The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type. Today, the use of window-oriented, point-andpick interfaces has reduced reliance on typed commands, but some power-users continue to prefer a command-oriented mode of interaction. A number of design

issues arise when typed commands or menu labels are provided as a mode of interaction:

• Will every menu option have a corresponding command?

• What form will commands take? Options include a control sequence (e.g., alt-P), function keys, or a typed word.

• How difficult will it be to learn and remember the commands? What can be done if a command is forgotten?

• Can commands be customized or abbreviated by the user?

• Are menu labels self-explanatory within the context of the interface?

• Are submenus consistent with the function implied by a master menu item? As I noted earlier in this chapter, conventions for command usage should be established across all applications. It is confusing and often error-prone for a user to type alt-D when a graphics object is to be duplicated in one application and alt-D when a graphics object is to be deleted in another. The potential for error is obvious.

Application accessibility.

As computing applications become ubiquitous, software engineers must ensure that interface design encompasses mechanisms that enable easy access for those with special needs. *Accessibility* for users (and software engineers) who may be physically challenged is an imperative for ethical, legal, and business reasons. A variety of accessibility guidelines (e.g.,

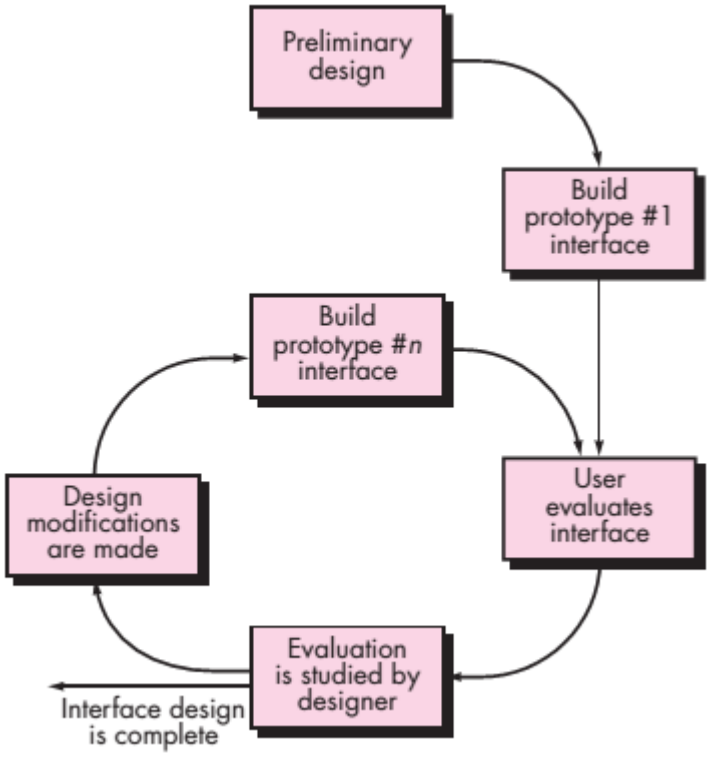
[W3C03])—many designed for Web applications but often applicable to all types of software—provide detailed suggestions for designing interfaces that achieve varying levels of accessibility. Others (e.g., [App08], [Mic08]) provide specific guidelines for "assistive technology" that addresses the needs of those with visual, hearing, mobility, speech, and learning impairments.

Internationalization. Software engineers and their managers invariably underestimate the effort and skills required to create user interfaces that accommodate the needs of different locales and languages. Too often, interfaces are designed for one locale and language and then jury-rigged to work in other countries. The challenge for interface designers is to create "globalized" software. That is, user interfaces should be designed to accommodate a generic core of functionality that can be delivered to all who use the software. *Localization* features enable the interface to be customized for a specific market.

A variety of internationalization guidelines (e.g., [IBM03]) are available to software engineers. These guidelines address broad design issues (e.g., screen layouts may differ in various markets) and discrete implementation issues (e.g., different alphabets may create specialized labeling and spacing requirements). The *Unicode* standard [Uni03] has been developed to address the daunting challenge of managing dozens of natural languages with hundreds of characters and symbols.

Design Evaluation. Once you create an operational user interface prototype, it must be evaluated to determine whether it meets the needs of the user. Evaluation can span a formality spectrum that ranges from an informal "test drive," in which a user provides impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end users.

The user interface evaluation cycle takes the form shown in Figure 11.5. After the design model has been completed, a first-level prototype is created. The prototype is evaluated by the user, who provides you with direct comments about the efficacy of the interface. In addition, if formal evaluation techniques are used (e.g., questionnaires, rating sheets), you can extract information from these data (e.g., 80 percent of all users did not like the mechanism for saving data files). Design modifications are made based on user input, and the next level prototype is created. The evaluation cycle continues until no further modifications to the interface design are necessary.



The interface design evaluation cycle

The prototyping approach is effective, but is it possible to evaluate the quality of a user interface before a prototype is built? If you identify and correct potential problems early, the number of loops through the evaluation cycle will be reduced and development time will shorten. If a design model of the interface has been created, a number of evaluation criteria [Mor81] can be applied during early design reviews:

1. The length and complexity of the requirements model or written specification of the system and its interface provide an indication of the amount of learning required by users of the system.

2. The number of user tasks specified and the average number of actions per task provide an indication of interaction time and the overall efficiency of the system.

3. The number of actions, tasks, and system states indicated by the design model imply the memory load on users of the system.

4. Interface style, help facilities, and error handling protocol provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.

Once the first prototype is built, you can collect a variety of qualitative and quantitative data that will assist in evaluating the interface. To collect qualitative data, questionnaires can be distributed to users of the prototype. Questions can be: (1) simple yes/no response, (2) numeric response, (3) scaled (subjective) response, (4) Likert scales (e.g., strongly agree, somewhat agree), (5) percentage (subjective) response, or (6) open-ended.

If quantitative data are desired, a form of time-study analysis can be conducted. Users are observed during interaction, and data—such as number of tasks correctly completed over a standard time period, frequency of actions, sequence of actions, time spent "looking" at the display, number and types of errors, error recovery time, time spent using help, and number of help references per standard time period—are collected and used as a guide for interface modification.

POSSIBLE QUESTIONS

PART - B

1. Explain the golden rules that guide the design user interface.
2. Illustrate the Interface analysis and design models and user interface design process in detail.
3. Explain the Task analysis and modeling approach in detail.
4. Explicate the various design models and framework activities in user interface design
5. Illustrate the important principles that guide for effective user interface design
6. Describe the interface design concepts in applying interface design steps.
7. Design the User Interfaces for the Student Information System.
8. Elucidate the User Interface Design Patterns and Design issues in detail.

KARPAGAM ACADEMY OF HIGHER EDUCATION					
CLASS: III BSC CS **COURSE NAME: SOFTWARE ENGINEERING** **COURSE CODE: 15CSU601** **UNIT: IV** **BATCH-2015-2018**					
ONE MARKS					
Questions	opt1	opt2	opt3	opt4	Answer
Interface design focuses on _____ areas of concern.	2	3	4	5	3
Frustration and _____ are part of daily life for many users of computerized information system	sadness	happiness	enjoyment	anxiety	anxiety
_____ creates effective communication medium between a human and a computer.	user interface design	architectural design	code design	procedure design	user interface design
_____ identifies interface objects and actions and then creates a screen layout that form the basis for a user interface prototype.	design	coding	testing	analysis	design
_____ begins with the identification of user, task and environmental requirements.	user interface design	architectural design	code design	procedure design	user interface design
There are _____ golden rules.	2	3	4	5	3
We should define interaction modes in a way that does not force a user into unnecessary or undesired actions.	interaction modes	interface constraints	design principles	design analysis	interaction modes
We should provide _____ interaction.	rigid	flexible	encouraging	enthusiastic	flexible
We should design for direct interaction with _____ that appear on the screen	code	class	objects	user	objects
We should hide technical _____ from the casual user	reactions	actions	internals	interactions	internals
We should streamline _____ as skill levels advance and allow the interaction to be customized.	internals	interaction	actions	reactions	interaction
We should allow user interaction to be _____ and undoable	interruptible	flexible	rigid	encouraging	interruptible
We should allow user interaction to interruptible and _____.	undoable	flexible	rigid	encouraging	undoable
We should define shortcuts that are _____.	encouraging	intuitive	default	past actions	intuitive
We should define _____ that are intuitive.	shortcuts	broad area	interruptible actions	interactions	shortcuts

We should disclose information in a _____ fashion.	open	progressive	streamline	flexible	progressive
The visual layout of the _____ should be based on a real world metaphor.	interaction modes	interface	design	structure	interface
The interface should present and acquire _____ in a consistent fashion.	information	task	knowledge	idea	information
The interface should present and acquire information in a _____ fashion.	consistent	inconsistent	rigid	flexible	consistent
A _____ of the entire system incorporates data, architectural interface, and procedural representations of the software	data model	design model	user model	system image	design model
The software engineer creates a _____.	design model	data model	interface model	system image	design model
The end user develops a mental image that is often called the _____.	design model	user model	data model	system image	user model
The implementers of the system create a _____.	design model	system image	data model	user model	system image
Users are categorized into _____ types.	2	3	4	5	3
Users with no syntactic knowledge of the system and little semantic knowledge of the application or computer usage are called _____.	knowledgeable intermittent users	knowledgeable frequent users	novices	all of the above	novices
Users with reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface are called _____.	novices	knowledgeable, intermittent users	knowledgeable, frequent users	all of the above	knowledgeable, intermittent users
Users with good semantic and syntactic knowledge that often leads to the "power-user syndrome" are called _____.	novices	knowledgeable, intermittent users	knowledgeable, frequent users	all of the above	knowledgeable, frequent users
Individuals who look for shortcuts and abbreviated modes of interaction are called _____.	novices	knowledgeable, intermittent users	knowledgeable, frequent users	Testers	knowledgeable, frequent users
The _____ is the image of the system that end-users carry in their heads.	user's model	data model	design model	system image	user's model
Stepwise elaboration is called _____.	functional decomposition	data abstraction	modularity	modular protection	functional decomposition

_____ is the only way that we can accurately translate a customer's requirements into a finished software product or system.	specification	design	data	prototype	design
Validation focuses on _____ criteria.	2	3	4	5	2
Task analysis can be applied in _____ ways.	2	3	4	5	3
Task analysis for interface design used _____ approach.	object oriented approach	top down approach	bottom up approach	all of the above	object oriented approach
The overall approach to task analysis, a human engineer must first _____ and classify tasks.	discuss	define	describe	list	define
There are _____ steps in interface design activities.	4	5	6	7	7
_____ refers to the deviation from average time.	system response time	variability	system mean time	all of the above	variability
System response time has _____ important characteristics.		3	4	5	2
A _____ is designed into the software from the beginning.	integrated help facility	system response time	variability	all of the above	integrated help facility
Component level design also called _____.	procedural abstraction	procedural design	stepwise refinement	decomposition	procedural design
_____ must be translated into operational software	data	architectural	interface design	all of the above	all of the above
A _____ performs component level design.	user	top level management	software engineer	middle level management	software engineer
The _____ represents the software in a way that allows one to review the details of the design for correctness and consistency with earlier design representations.	component level design	procedural design	data design	data design	component level design
Design, representations of data, architecture, and interfaces form the foundation for _____.	procedural design	component level design	data design	code design	component level design

_____ notation is used to represent the design.	graphical	tabular	text-based	all of the above	graphical
Any program, regardless of application area or technical complexity, can be designed and implemented using only the _____ structured constructs.	2	3	4	5	3
A box in a flowchart is used to indicate a _____.	processing step	logical condition	flow of control	start	processing step
A diamond in a flowchart is used to indicate a _____.	processing step	logical condition	flow of control	start	logical condition
The arrows in a flowchart is used to indicate a _____.	processing step	logical condition	flow of control	start	flow of control
A picture is worth a _____ words.	100	1000	10000	100000	1000
The following construct is fundamental to structured programming.	sequence	condition	repetition	all of the above	all of the above
_____ implements processing steps that are essential in the specification of any algorithm.	sequence	condition	repetition	selection	sequence
_____ provides the facility for selected processing steps that are essential in the specification of any algorithm	sequence	condition	repetition	selection	condition
_____ allows for looping.	sequence	condition	repetition	selection	repetition
Another graphical design tool, the _____ evolved from a desire to develop a procedural design representation that would not allow violation of the structured constructs.	box diagram	flowchart	transition diagram	decision table	box diagram
PDL is the abbreviation of _____.	Process Design Language	Program Design Language	Program Document Language	Program Document Language	Program Design Language
A design language should have the _____ characters.	2	3	4	5	4
Design notation should support the development of modular software and provide a means for interface specification. This attribute of design notation is called _____.	modularity	simplicity	ease of editing	maintainability	modularity
Design notation should be relatively simple to learn, relatively easy to use, and generally easy to read. This attribute of the design notation is called _____.	modularity	simplicity	ease of editing	maintainability	simplicity

The procedural design may require modification as the software process proceeds. The ease with which a design representation can be edited can help facilitate each software engineering task is called _____.	modularity	simplicity	ease of editing	maintainability	ease of editing

UNIT-V

SYLLABUS

Testing Tactics: Software Testing Fundamentals- Black -Box and White-Box Testing- White Box Testing-Basis Path Testing- Control Structure Testing: Condition Testing- Data Flow Testing-Loop Testing- Black Box Testing- Quality Concepts: Quality- Quality Control –Quality Assurance –Cost Of Quality.

TESTING TACTICS

Testing presents an interesting anomaly for software engineers, who by their nature are constructive people. Testing requires that the developer discard preconceived notions of the "correctness" of software just developed and then work hard to design test cases to "break" the software. Beizer describes this situation effectively when he states:

There's a myth that if we were really good at programming, there would be no bugs to catch. If only we could really concentrate, if only everyone used structured programming, top-down design, . . . then there would be no bugs. So goes the myth.

There are bugs, the myth says, because we are bad at what we do; and if we are bad at it, we should feel guilty about it. Therefore, testing and test case design is an admission of failure, which instills a goodly dose of guilt.

And the tedium of testing is just punishment for our errors. Punishment for what? For being human? Guilt for what? For failing to achieve inhuman perfection? For not distinguishing between what another programmer thinks and what he says? For failing to be telepathic? For not solving human communications problems that have been kicked around . . . for forty centuries? Should testing instill guilt? Is testing really destructive? The answer to these questions is "No!"

SOFTWARE TESTING FUNDAMENTALS

The goal of testing is to find errors, and a good test is one that has a high probability of finding an error. Therefore, you should design and implement a computerbased system or a product with "testability" in mind. At the same time, the tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.

Testability. James Bach provides the following definition for testability: "*Software testability* is simply how easily [a computer program] can be tested." The following characteristics lead to testable software.

Operability. "The better it works, the more efficiently it can be tested." If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests, allowing testing to progress without fits and starts.

Observability. "What you see is what you test." Inputs provided as part of testing produce distinct outputs. System states and variables are visible or queriable during execution. Incorrect output is easily identified. Internal errors are automatically detected and reported. Source code is accessible.

Controllability. "The better we can control the software, the more the testing can be automated and optimized." All possible outputs can be generated through some combination of input, and I/O formats are consistent and structured. All code is executable through some combination of input. Software and hardware states and variables can be controlled directly by the test engineer. Tests can be conveniently specified, automated, and reproduced.

Decomposability. "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting." The software system is built from independent modules that can be tested independently.

Simplicity. "The less there is to test, the more quickly we can test it." The program should exhibit *functional simplicity* (e.g., the feature set is the minimum necessary to meet requirements); *structural simplicity* (e.g., architecture is modularized to limit the propagation of faults), and *code simplicity* (e.g., a coding standard is adopted for ease of inspection and maintenance).

Stability. "The fewer the changes, the fewer the disruptions to testing." Changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests. The software recovers well from failures.

Understandability. "The more information we have, the smarter we will test." The architectural design and the dependencies between internal, external, and shared components are well understood. Technical documentation is instantly accessible, well organized, specific and detailed, and accurate. Changes to the design are communicated to testers.

You can use the attributes suggested by Bach to develop a software configuration (i.e., programs, data, and documents) that is amenable to testing.

Test Characteristics.

And what about the tests themselves? Kaner, Falk, and Nguyen [Kan93] suggest the following attributes of a "good" test:

A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail. Ideally, the classes of failure are probed. For example, one class of potential failure in a graphical user interface is the failure to recognize proper mouse position. A set of tests would be designed to exercise the mouse in an attempt to demonstrate an error in mouse position recognition.

A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different).

A good test should be "best of breed" [Kan93]. In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

BLACK -BOX AND WHITE-BOX TESTING

Any engineered product (and most other things) can be tested in one of two ways:

(1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function.

(2) Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised. The first test approach takes an external view and is called black-box testing. The second requires an internal view and is termed white-box testing.

Black-box testing alludes to tests that are conducted at the software interface. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software. *White-box testing* of software is predicated on close examination of procedural detail. Logical paths through the software and collaborations between components are tested by exercising specific sets of conditions and/or loops.

At first glance it would seem that very thorough white-box testing would lead to "100 percent correct programs." All we need do is define all logical paths, develop test cases to exercise them, and evaluate results, that is, generate test cases to exercise program logic

exhaustively. Unfortunately, exhaustive testing presents certain logistical problems. For even small programs, the number of possible logical paths can be very large. White-box testing should not, however, be dismissed as impractical. A limited number of important logical paths can be selected and exercised. Important data structures can be probed for validity.

White-box testing, sometimes called *glass-box testing,* is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases. Using white-box testing methods, you can derive test cases that (1) guarantee that all independent paths within a module have been exercised at least once, (2) exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.

WHITE BOX TESTING

White-box testing, sometimes called *glass-box testing,* is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases. Using white-box testing methods, you can derive test cases that

(1) guarantee that all independent paths within a module have been exercised at least once,

(2) exercise all logical decisions on their true and false sides,

(3) execute all loops at their boundaries and within their operational bounds, and

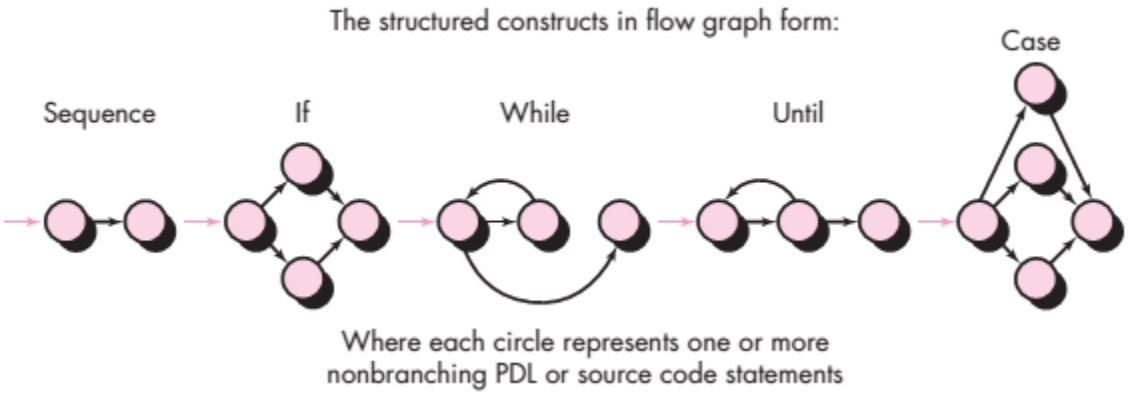
(4) exercise internal data structures to ensure their validity.

BASIS PATH TESTING

Basis path testing is a white-box testing technique first proposed by Tom McCabe [McC76]. The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

Flow Graph Notation

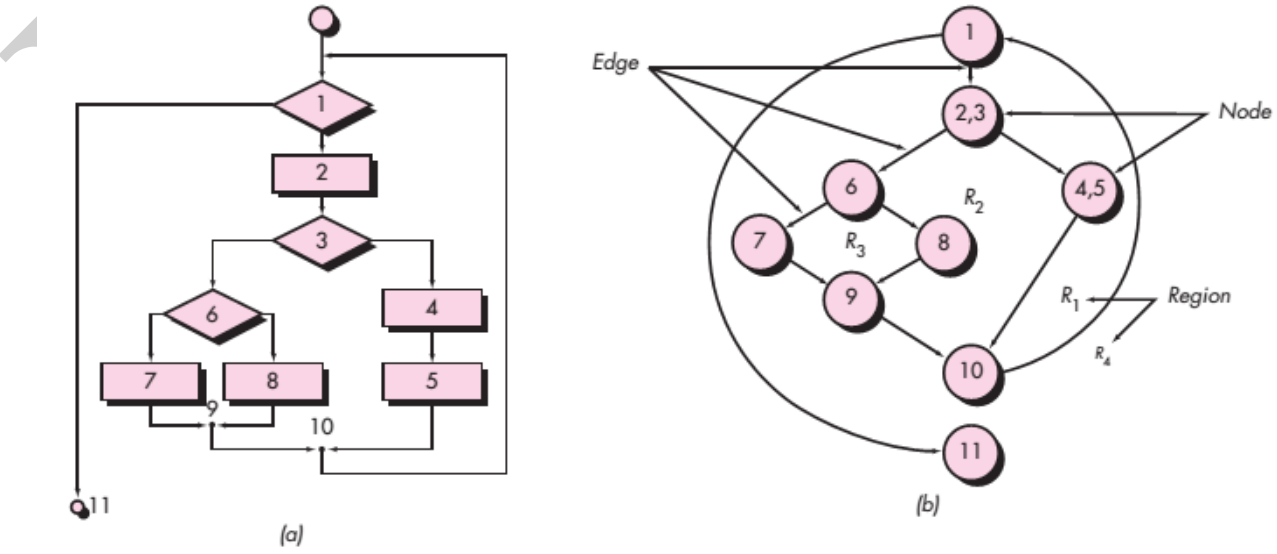
Before we consider the basis path method, a simple notation for the representation of control flow, called a *flow graph* (or *program graph*) must be introduced. The flow graph depicts logical control flow using the notation illustrated in Figure. Each structured construct has a corresponding flow graph symbol.



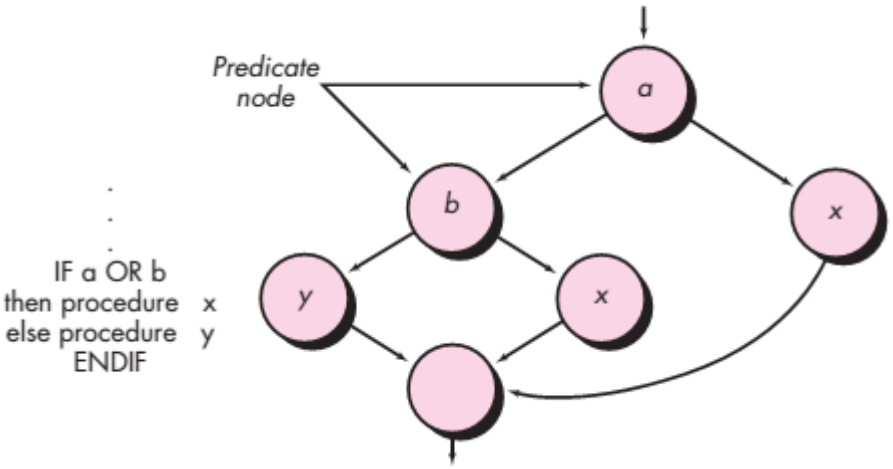
Flow graph notation

To illustrate the use of a flow graph, consider the procedural design representation in Figure. Here, a flowchart is used to depict program control structure. Figure 18.2b maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart). Referring to

Figure, each circle, called a *flow graph node*, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called *edges* or *links*, represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the flow graph symbol for the if-then-else construct). Areas bounded by edges and nodes are called *regions*. When counting regions, we include the area outside the graph as a region.4



(a) Flowchart and (b) flow graph



Compound logic

When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure 18.3, the program design language (PDL) segment translates into the flow graph shown. Note that a separate node is created for each of the conditions *a* and *b* in the statement IF *a* OR *b*. Each node that contains a condition is called a *predicate node* and is characterized by two or more edges emanating from it.

Independent Program Paths

An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure 18.2b is Path 1: 1-11 Path 2: 1-2-3-4-5-10-1-11 Path 3: 1-2-3-6-8-9-10-1-11 Path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge. The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Paths 1 through 4 constitute a *basis set* for the flow graph in Figure 18.2b. That is, if you can design tests to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

How do you know how many paths to look for? The computation of cyclomatic complexity provides the answer. *Cyclomatic complexity* is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides you with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

Cyclomatic complexity has a foundation in graph theory and provides you with an extremely useful software metric. Complexity is computed in one of three ways:

1. The number of regions of the flow graph corresponds to the cyclomatic complexity.

2. Cyclomatic complexity $V(G)$ for a flow graph G is defined as $V(G)$ E N 2 where E is the number of flow graph edges and N is the number of flow graph nodes.

3. Cyclomatic complexity $V(G)$ for a flow graph G is also defined as $V(G)$ P 1 where P is the number of predicate nodes contained in the flow graph G. Referring once more to the flow graph in Figure 18.2b, the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions. **2.** $V(G)$ 11 edges 9 nodes 2 4. **3.** $V(G)$ 3 predicate nodes 1 4.

Therefore, the cyclomatic complexity of the flow graph in Figure 18.2b is 4. More important, the value for $V(G)$ provides you with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

Deriving Test Cases The basis path testing method can be applied to a procedural design or to source code. In this section, I present basis path testing as a series of steps. The procedure *average*, depicted in PDL in Figure 18.4, will be used as an example to illustrate each step in the test-case design method. Note that *average*, although an extremely simple algorithm, contains compound conditions and loops. The following steps can be applied to derive the basis set:

1. Using the design or code as a foundation, draw a corresponding flow graph. A flow graph is created using the symbols and construction rules presented in Section 18.4.1. Referring to the PDL for *average* in Figure 18.4, a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes. The corresponding flow graph is shown in Figure 18.5.

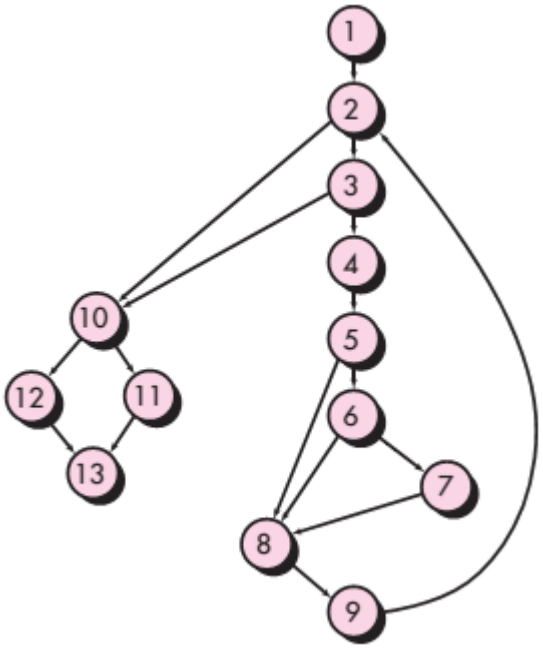
2. Determine the cyclomatic complexity of the resultant flow graph.

The cyclomatic complexity *V*(*G*) is determined by applying the algorithms described in Section 18.4.2. It should be noted that *V*(*G*) can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure *average*, compound conditions count as two) and adding 1. Referring to Figure 18.5, *V*(*G*) 6 regions *V*(*G*) 17 edges 13 nodes 2 6 *V*(*G*) 5 predicate nodes 1 6

```
PROCEDURE average;

    * This procedure computes the average of 100 or fewer
      numbers that lie between bounding values; it also computes the
      sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
       minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
1   i = 1;
    total.input = total.valid = 0;          2
    sum = 0;
    DO WHILE value[i] <> -999 AND total.input < 100   3
4      increment total.input by 1;
       IF value[i] > = minimum AND value[i] < = maximum   6
5         THEN increment total.valid by 1;
7                sum = s sum + value[i]
          ELSE skip
8      ENDIF
       increment i by 1;
9   ENDDO
    IF total.valid > 0    10
11     THEN average = sum / total.valid;
12     ELSE average = -999;
13  ENDIF
END average
```

PDL with nodes identified

3. Determine a basis set of linearly independent paths. The value of *V*(*G*) provides the upper bound on the number of linearly independent paths through the program control structure. In the case of procedure *average*, we expect to specify six paths: Path 1: 1-2-10-11-13 Path 2: 1-2-10-12-13



Flow graph for the procedure *average*

Path 3: 1-2-3-10-11-13 Path 4: 1-2-3-4-5-8-9-2-. . . Path 5: 1-2-3-4-5-6-8-9-2-. . . Path 6: 1-2-3-4-5-6-7-8-9-2-. . . The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable. It is often worthwhile to identify predicate nodes as an aid in the derivation of test cases. In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes.

4. Prepare test cases that will force execution of each path in the basis set.

Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

It is important to note that some independent paths (e.g., path 1 in our example) cannot be tested in stand-alone fashion. That is, the combination of data required to traverse the path cannot be achieved in the normal flow of the program. In such cases, these paths are tested as part of another path test.

Graph Matrices

The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. A data structure, called a *graph matrix,* can be quite useful for developing a software tool that assists in basis path testing. A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes. A simple example of a flow graph and its corresponding graph matrix [Bei90] is shown in Figure 18.6.

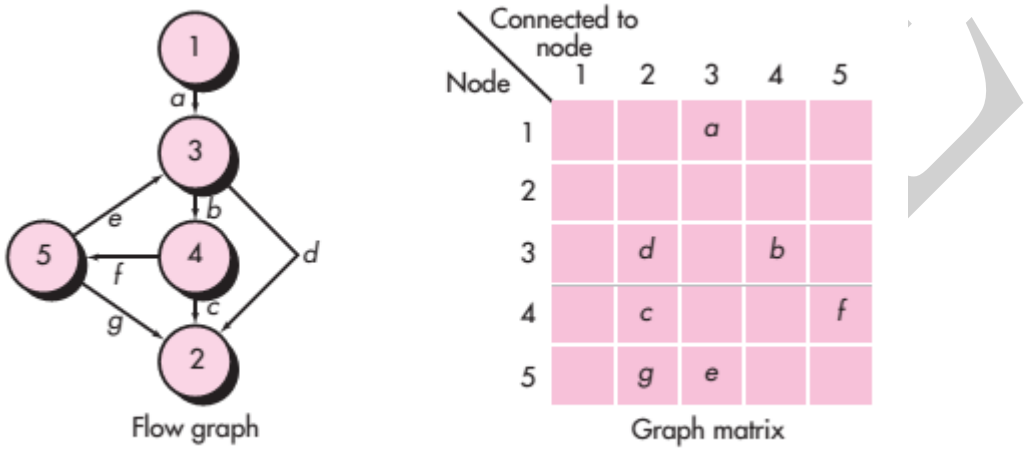


Fig Graph matrix

Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge *b*.

To this point, the graph matrix is nothing more than a tabular representation of a flow graph. However, by adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing.

The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But link weights can be assigned other, more interesting properties:

• The probability that a link (edge) will be execute. • The processing time expended during traversal of a link • The memory required during traversal of a link • The resources required during traversal of a link.

Beizer [Bei90] provides a thorough treatment of additional mathematical algorithms that can be applied to graph matrices. Using these techniques, the analysis required to design test cases can be partially or fully automated.

CONTROL STRUCTURE TESTING

The basis path testing technique described in Section 18.4 is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. In this section, other variations on control structure testing are discussed. These broaden testing coverage and improve the quality of white-box testing.

CONDITION TESTING

Condition testing [Tai89] is a test-case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form *E* 1 <relational-operator> *E*2 where *E*1 and *E*2 are arithmetic expressions and <relational-operator> is one of the following: , , , (nonequality), , or . A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (), AND (&), and NOT (¬). A condition without relational expressions is referred to as a Boolean expression.

If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include Boolean operator errors (incorrect/missing/extra Boolean operators), Boolean variable errors, Boolean parenthesis errors, relational operator errors, and arithmetic expression errors. The condition testing method focuses on testing each condition in the program to ensure that it does not contain errors.

DATA FLOW TESTING

The data flow testing method [Fra93] selects test paths of a program according to the locations of definitions and uses of variables in the program. To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with *S* as its statement number,

DEF(*S*) {*X* | statement *S* contains a definition of *X*} USE(*S*) {*X* | statement *S* contains a use of *X*}

If statement *S* is an *if* or *loop statement*, its DEF set is empty and its USE set is based on the condition of statement *S*. The definition of variable *X* at statement *S* is said to be *live* at statement *S'* if there exists a path from statement *S* to statement *S'* that contains no other definition of *X*.

A *definition-use (DU) chain* of variable *X* is of the form [*X, S, S'*], where *S* and *S'* are statement numbers, *X* is in DEF(*S*) and USE(*S'*), and the definition of *X* in statement *S* is live at statement *S'*.

One simple data flow testing strategy is to require that every DU chain be covered at least once. We refer to this strategy as the DU testing strategy. It has been shown that DU testing does not guarantee the coverage of all branches of a program. However, a branch is not guaranteed to be covered by DU testing only in rare situations such as if-then-else constructs in which the *then part* has no definition of any variable and the *else part* does not exist. In this situation, the else branch of the *if* statement is not necessarily covered by DU testing.

LOOP TESTING

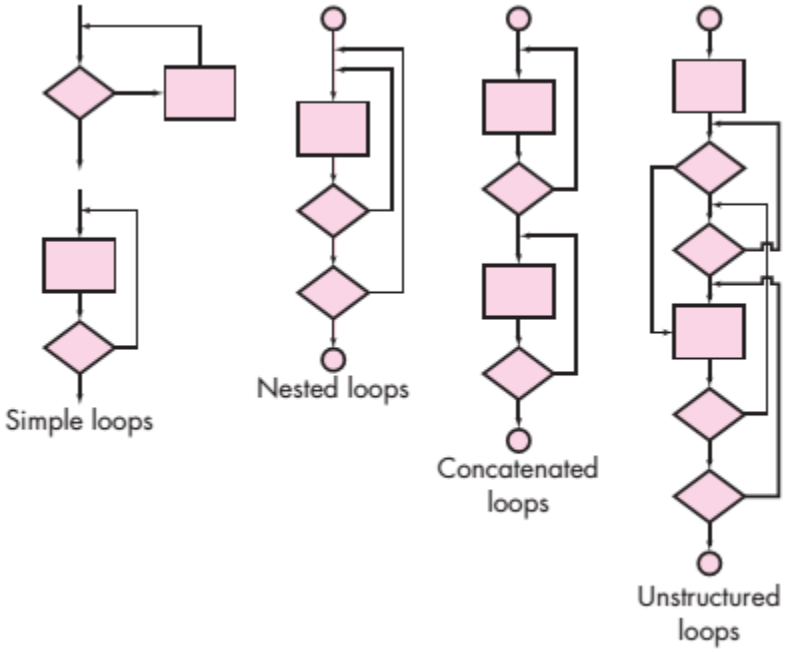
Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests. *Loop testing* is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops [Bei90] can be defined: simple loops, concatenated loops, nested loops, and unstructured loops (Figure 18.7).

Simple loops. The following set of tests can be applied to simple loops, where *n* is the maximum number of allowable passes through the loop.

1. Skip the loop entirely.

2. Only one pass through the loop.

3. Two passes through the loop.



Classes of Loops

4. *m* passes through the loop where *m n.*

5. *n* 1, *n*, *n* 1 passes through the loop.

Nested loops. If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Beizer [Bei90] suggests an approach that will help to reduce the number of tests:

1. Start at the innermost loop. Set all other loops to minimum values.

2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.

3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.

4. Continue until all loops have been tested.

Concatenated loops. Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

Unstructured loops. Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs (Chapter 10).

BLACK BOX TESTING

Black-box testing, also called *behavioral testing*, focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program.

Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than whitebox methods. Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external database access, (4) behavior or performance errors, and (5) initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, blackbox testing tends to be applied during later stages of testing (see Chapter 17). Because black-box testing purposely disregards control structure, attention is focused on the information domain. Tests are designed to answer the following questions:

- How is functional validity tested?
- How are system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation? By applying black-box techniques, you derive a set of test cases that satisfy the following criteria [Mye79]: (1) test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing, and (2) test cases that tell you something

about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

Graph-Based Testing Methods The first step in black-box testing is to understand the objects5 that are modeled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify "all objects have the expected relationship to one another" [Bei95]. Stated in another way, software testing begins by creating a graph of important objects and their relationships and

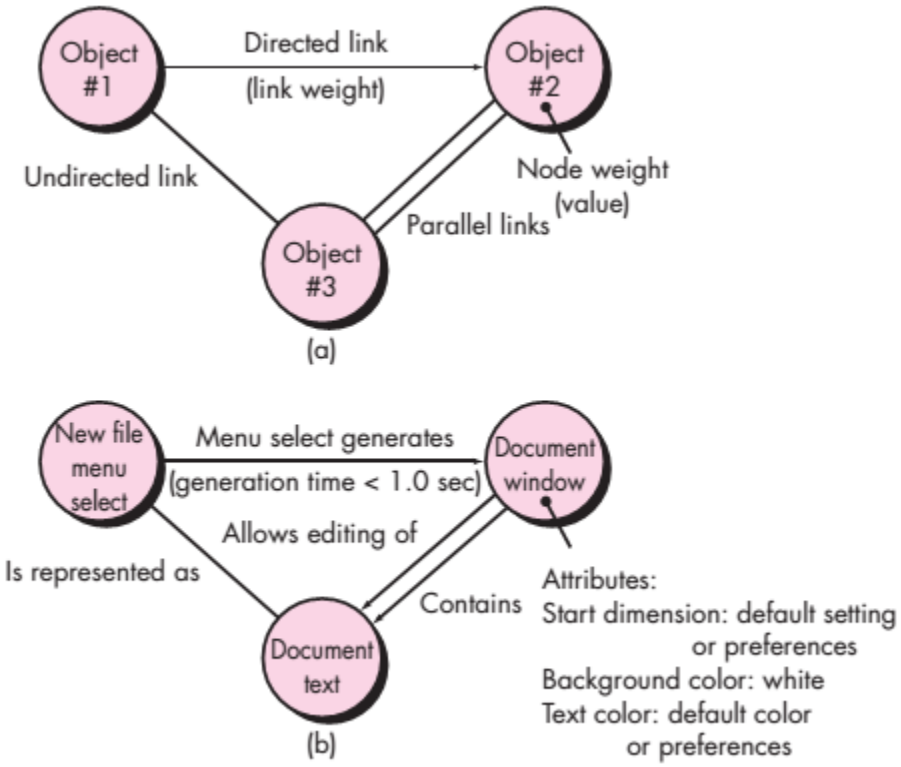


Fig (a) Graph notation; (b) simple example

then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

To accomplish these steps, you begin by creating a *graph*—a collection of *nodes* that represent objects, *links* that represent the relationships between objects, *node weights* that describe the properties of a node (e.g., a specific data value or state behavior), and *link weights* that describe some characteristic of a link. The symbolic representation of a graph is shown in Figure 18.8a. Nodes are represented as circles connected by links that take a number of different forms.

A *directed link* (represented by an arrow) indicates that a relationship moves in only one direction. A *bidirectional link,* also called a *symmetric link,* implies that the relationship applies in both directions. *Parallel links* are used when a number of different relationships are established between graph nodes.

As a simple example, consider a portion of a graph for a word-processing application where *Object #1* **newFile** (menu selection) *Object #2* **documentWindow** *Object #3* **documentText**

Referring to the figure, a menu select on **newFile** generates a document window. The node weight of **documentWindow** provides a list of the window attributes that are to be expected when the window is generated. The link weight indicates that the window must be generated in less than 1.0 second. An undirected link establishes a symmetric relationship between the **newFile** menu selection and **documentText,** and parallel links indicate relationships between **documentWindow** and **documentText.** In reality, a far more detailed graph would have to be generated as a precursor to test-case design. You can then derive test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships. Beizer [Bei95] describes a number of behavioral testing methods that can make use of graphs:

Transaction flow modeling. The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an online service), and the links represent the logical connection between steps (e.g., **flightInformationInput** is followed by *validationAvailabilityProcessing*). The data flow diagram (Chapter 7) can be used to assist in creating graphs of this type.

Finite state modeling. The nodes represent different user-observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state (e.g., **orderInformation** is verified during *inventoryAvailabilityLook-up* and is followed by **customerBillingInformation** input). The state diagram (Chapter 7) can be used to assist in creating graphs of this type.

Data flow modeling. The nodes are data objects, and the links are the transformations that occur to translate one data object into another. For example, the node FICA tax withheld (**FTW**) is computed from gross wages (**GW**) using the relationship, **FTW 0.62 GW.**

Timing modeling. The nodes are program objects, and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes. A detailed discussion of each of these graph-based testing methods is beyond the scope of this book. If you have further interest, see [Bei95] for a comprehensive coverage.

Equivalence Partitioning *Equivalence partitioning* is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many test cases to be executed before the general error is observed. Test-case design for equivalence partitioning is based on an evaluation of *equivalence classes* for an input condition. Using concepts introduced in the preceding section, if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present [Bei95]. An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.

2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.

3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.

4. If an input condition is Boolean, one valid and one invalid class are defined. By applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

Boundary Value Analysis A greater number of errors occurs at the boundaries of the input domain rather than in the "center." It is for this reason that *boundary value analysis* (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values.

Boundary value analysis is a test-case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well [Mye79].

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a range bounded by values *a* and *b*, test cases should be designed with values *a* and *b* and just above and just below *a* and *b*.

2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.

3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.

4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

Orthogonal Array Testing

There are many applications in which the input domain is relatively limited. That is, the number of input parameters is small and the values that each of the parameters may take are clearly bounded. When these numbers are very small (e.g., three input parameters taking on three discrete values each), it is possible to consider every input permutation and exhaustively test the input domain. However, as the number of input values grows and the number of discrete values for each data item increases, exhaustive testing becomes impractical or impossible.

Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding *region faults*—an error category associated with faulty logic within a software component. To illustrate the difference between orthogonal array testing and more conventional "one input item at a time" approaches, consider a system that has three input items, *X*, *Y*, and *Z*. Each of these input items has three discrete values associated with it. There are 33 27 possible test cases. Phadke [Pha97] suggests a geometric view of the possible test cases associated with X, Y, and Z illustrated in Figure 18.9.

Referring to the figure, one input item at a time may be varied in sequence along each input axis. This results in relatively limited coverage of the input domain (represented by the left-hand cube in the figure).

When orthogonal array testing occurs, an L9 *orthogonal array* of test cases is created. The L9 orthogonal array has a "balancing property" [Pha97]. That is, test cases (represented by dark dots in the figure) are "dispersed uniformly throughout the test domain," as illustrated in the right-hand cube in Figure 18.9. Test coverage across the input domain is more complete.

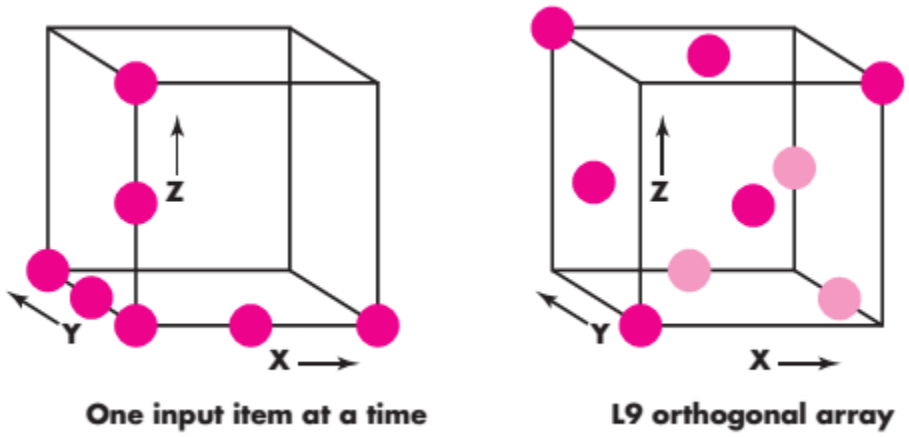


Fig A geometric view of test cases

To illustrate the use of the L9 orthogonal array, consider the *send* function for a fax application. Four parameters, P1, P2, P3, and P4, are passed to the *send* function. Each takes on three discrete values. For example, P1 takes on values:

P1 1, send it now

P1 2, send it one hour later

P1 3, send it after midnight

P2, P3, and P4 would also take on values of 1, 2, and 3, signifying other send functions.

If a "one input item at a time" testing strategy were chosen, the following sequence of tests (P1, P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3). Phadke [Pha97] assesses these test cases by stating:

Such test cases are useful only when one is certain that these test parameters do not interact. They can detect logic faults where a single parameter value makes the software malfunction. These faults are called *single mode faults.* This method cannot detect logic faults that cause malfunction when two or more parameters simultaneously take certain values; that is, it cannot detect any interactions. Thus its ability to detect faults is limited.

Given the relatively small number of input parameters and discrete values, exhaustive testing is possible. The number of tests required is 34 81, large but manageable. All faults associated with data item permutation would be found, but the effort required is relatively high.

The orthogonal array testing approach enables you to provide good test coverage with far fewer test cases than the exhaustive strategy. An L9 orthogonal array for the fax *send* function is illustrated in Figure 18.10.

Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Fig 18.10. An L9 orthogonal array

Phadke [Pha97] assesses the result of tests using the L9 orthogonal array in the following manner:

Detect and isolate all single mode faults. A single mode fault is a consistent problem with any level of any single parameter. For example, if all test cases of factor P1 1 cause an error condition, it is a single mode failure. In this example tests 1, 2 and 3 [Figure 18.10] will show errors. By analyzing the information about which tests show errors, one can identify which parameter values cause the fault. In this example, by noting that tests 1, 2, and 3 cause an error, one can isolate [logical processing associated with "send it now" (P1 1)] as the source of the error. Such an isolation of fault is important to fix the fault.

Detect all double mode faults. If there exists a consistent problem when specific levels of two parameters occur together, it is called a *double mode fault.* Indeed, a double mode fault is an indication of pairwise incompatibility or harmful interactions between two test parameters.

Multimode faults. Orthogonal arrays [of the type shown] can assure the detection of only single and double mode faults. However, many multimode faults are also detected by these tests. You can find a detailed discussion of orthogonal array testing in [Pha89].

QUALITY CONCEPTS:

The drumbeat for improved software quality began in earnest as software became increasingly integrated in every facet of our lives. By the 1990s, major corporations recognized that billions of dollars each year were being wasted on software that didn't deliver the features and functionality that were promised.

Worse, both government and industry became increasingly concerned that a major software fault might cripple important infrastructure, costing tens of billions more. By the turn of the century, *CIO Magazine* [Lev01] trumpeted the headline, "Let's Stop Wasting $78 Billion a Year," lamenting the fact that "American businesses spend billions for software that doesn't do what it's supposed to do." *InformationWeek* [Ric01] echoed the same concern:

Despite good intentions, defective code remains the hobgoblin of the software industry, accounting for as much as 45% of computer-system downtime and costing U.S. companies about $100 billion last year in lost productivity and repairs, says the Standish Group, a market research firm. That doesn't include the cost of losing angry customers. Because IT shops write applications that rely on packaged infrastructure software, bad code can wreak havoc on custom apps as well. . . .

Just how bad is bad software? Definitions vary, but experts say it takes only three or four defects per 1,000 lines of code to make a program perform poorly. Factor in that most programmers inject about one error for every 10 lines of code they write, multiply that by the millions of lines of code in many commercial products, then figure it costs software vendors at least half their development budgets to fix errors while testing. Get the picture?

In 2005, *ComputerWorld* [Hil05] lamented that "bad software plagues nearly every organization that uses computers, causing lost work hours during computer downtime, lost or corrupted data, missed sales opportunities, high IT support and maintenance costs, and low customer satisfaction. A year later, *InfoWorld* [Fos06] wrote about the "the sorry state of software quality" reporting that the quality problem had not gotten any better.

Today, software quality remains an issue, but who is to blame? Customers blame developers, arguing that sloppy practices lead to low-quality software. Developers blame customers (and other stakeholders), arguing that irrational delivery dates and a continuing stream of changes force them to deliver software before it has been fully validated. Who's right? *Both*—and that's the problem. In this chapter, I consider software quality as a concept and examine why it's worthy of serious consideration whenever software engineering practices are applied.

QUALITY

In his mystical book, *Zen and the Art of Motorcycle Maintenance*, Robert Persig [Per74] commented on the thing we call *quality*:

Quality . . . you know what it is, yet you don't know what it is. But that's self-contradictory. But some things are better than others; that is, they have more quality. But when you try to say what the quality is, apart from the things that have it, it all goes poof! There's nothing to talk about. But if you can't say what Quality is, how do you know what it is, or how do you know that it even exists? If no one knows what it is, then for all practical purposes it doesn't exist at all. But for all practical purposes it really does exist. What else are the grades based on? Why else would people pay fortunes for some things and throw others in the trash pile? Obviously some things are better than others . . . but what's the betterness? . . .

So round and round you go, spinning mental wheels and nowhere finding anyplace to get traction. What the hell is Quality? What is it?

Indeed—what is it?

At a somewhat more pragmatic level, David Garvin [Gar84] of the Harvard Business School suggests that "quality is a complex and multifaceted concept" that can be described from five different points of view. The *transcendental view* argues (like Persig) that quality is something that you immediately recognize, but cannot explicitly define. The *user view* sees

quality in terms of an end user's specific goals. If a product meets those goals, it exhibits quality. The *manufacturer's view* defines quality in terms of the original specification of the product. If the product conforms to the spec, it exhibits quality. The *product view* suggests that quality can be tied to inherent characteristics (e.g., functions and features) of a product. Finally, the *value-based view* measures quality based on how much a customer is willing to pay for a product. In reality, quality encompasses all of these views and more.

Quality of design refers to the characteristics that designers specify for a product. The grade of materials, tolerances, and performance specifications all contribute to the quality of design. As higher-grade materials are used, tighter tolerances and greater levels of performance are specified, the design quality of a product increases, if the product is manufactured according to specifications. In software development, quality of design encompasses the degree to which the design meets the functions and features specified in the requirements model. *Quality of conformance* focuses on the degree to which the implementation follows the design and the resulting system meets its requirements and performance goals.

But are quality of design and quality of conformance the only issues that software engineers must consider? Robert Glass [Gla98] argues that a more "intuitive" relationship is in order: user satisfaction compliant product good quality delivery within budget and schedule

At the bottom line, Glass contends that quality is important, but if the user isn't satisfied, nothing else really matters. DeMarco [DeM98] reinforces this view when he states: "A product's quality is a function of how much it changes the world for the better." This view of quality contends that if a software product provides substantial benefit to its end users, they may be willing to tolerate occasional reliability or performance problems.

SOFTWARE QUALITY

Even the most jaded software developers will agree that high-quality software is an important goal. But how do we define *software* quality? In the most general sense, software quality can be defined1 as: *An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.*

There is little question that the preceding definition could be modified or extended and debated endlessly. For the purposes of this book, the definition serves to emphasize three important points:

1. An *effective software process* establishes the infrastructure that supports any effort at building a high-quality software product. The management aspects of process create the checks and balances that help avoid project chaos—a key contributor to poor quality. Software engineering practices allow the developer to analyze the problem and design a solid solution—both critical to building high-quality software. Finally, umbrella activities such as change

management and technical reviews have as much to do with quality as any other part of software engineering practice.

2. A *useful product* delivers the content, functions, and features that the end user desires, but as important, it delivers these assets in a reliable, error-free

way. A useful product always satisfies those requirements that have been explicitly stated by stakeholders. In addition, it satisfies a set of implicit requirements (e.g., ease of use) that are expected of all high-quality software.

3. By *adding value for both the producer and user* of a software product, high quality software provides benefits for the software organization and the end user community. The software organization gains added value because high-quality software requires less maintenance effort, fewer bug fixes, and reduced customer support. This enables software engineers to spend more time creating new applications and less on rework. The user community gains added value because the application provides a useful capability in a way that expedites some business process. The end result is (1) greater software product revenue, (2) better profitability when an application supports a business process, and/or (3) improved availability of information that is crucial for the business.

QUALITY CONTROL

Quality control encompasses a set of software engineering actions that help to ensure that each work product meets its quality goals. Models are reviewed to ensure that they are complete and consistent. Code may be inspected in order to uncover and correct errors before testing commences. A series of testing steps is applied to uncover errors in processing logic, data manipulation, and interface communication. A combination of measurement and feedback allows a software team to tune the process when any of these work products fail to meet quality goals.

QUALITY ASSURANCE

Quality assurance establishes the infrastructure that supports solid software engineering methods, rational project management, and quality control actions—all pivotal if you intend to build high-quality software. In addition, quality assurance consists of a set of auditing and reporting functions that assess the effectiveness and completeness of quality control actions. The goal of quality assurance is to provide management and technical staff with the data necessary to be informed about product quality, thereby gaining insight and confidence that actions to achieve product quality are working. Of course, if the data provided through quality assurance identifies problems, it is management's responsibility to address the problems and apply the necessary resources to resolve quality issues.

COST OF QUALITY

The argument goes something like this—*we know that quality is important, but it costs us time and money—too much time and money to get the level of software quality we really want.* On its face, this argument seems reasonable (see Meyer's comments earlier in this section). There is no question that quality has a cost, but lack of quality also has a cost—not only to end users who must live with buggy software, but also to the software organization that has built and must maintain it. The real question is this: *which cost should we be worried about?* To answer this question, you must understand both the cost of achieving quality and the cost of low-quality software. The *cost of quality* includes all costs incurred in the pursuit of quality or in performing quality-related activities and the downstream costs of lack of quality. To understand these costs, an organization must collect metrics to provide a baseline for the current cost of quality, identify opportunities for reducing these costs, and provide a normalized basis of comparison. The cost of quality can be divided into costs associated with prevention, appraisal, and failure.

Prevention costs include (1) the cost of management activities required to plan and coordinate all quality control and quality assurance activities, (2) the cost of added technical activities to develop complete requirements and design models, (3) test planning costs, and (4) the cost of all training associated with these activities. *Appraisal costs* include activities to gain insight into product condition the "first time through" each process. Examples of appraisal costs include:

• Cost of conducting technical reviews (Chapter 15) for software engineering work products

• Cost of data collection and metrics evaluation (Chapter 23)

• Cost of testing and debugging (Chapters 18 through 21) *Failure costs* are those that would disappear if no errors appeared before or after shipping a product to customers. Failure costs may be subdivided into internal failure costs and external failure costs. *Internal failure costs* are incurred when you detect an error in a product prior to shipment. Internal failure costs include

• Cost required to perform rework (repair) to correct an error

• Cost that occurs when rework inadvertently generates side effects that must be mitigated

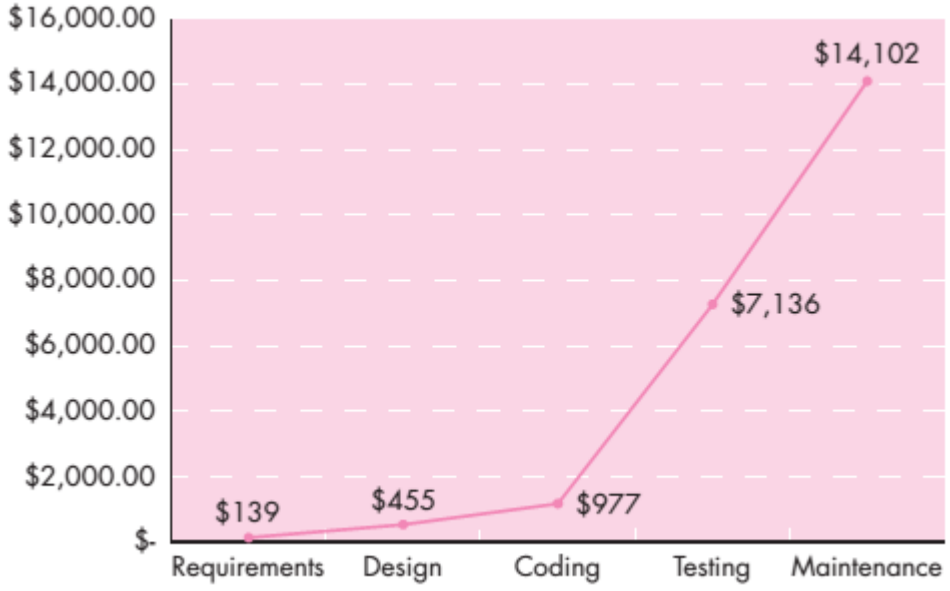
• Costs associated with the collection of quality metrics that allow an organization to assess the modes of failure

External failure costs are associated with defects found after the product has been shipped to the customer. Examples of external failure costs are complaint resolution, product return and replacement, help line support, and labor costs associated with warranty work. A poor reputation and the resulting loss of business is another external failure cost that is difficult to quantify but nonetheless very real. Bad things happen when low-quality software is produced.

In an indictment of software developers who refuse to consider external failure costs, Cem Kaner [Kan95] states:

Many of the external failure costs, such as goodwill, are difficult to quantify, and many companies therefore ignore them when calculating their cost-benefit tradeoffs. Other external failure costs can be reduced (e.g. by providing cheaper, lower-quality, post-sale support, or by charging customers for support) without increasing customer satisfaction. By ignoring the costs to our customers of bad products, quality engineers encourage quality-related decision-making that victimizes our customers, rather than delighting them.

As expected, the relative costs to find and repair an error or defect increase dramatically as we go from prevention to detection to internal failure to external failure costs. Figure 14.2, based on data collected by Boehm and Basili [Boe01b] and illustrated by Cigital Inc. [Cig07], illustrates this phenomenon. The industry average cost to correct a defect during code generation is approximately $977 per error. The industry average cost to correct the same error if it is



Relative cost of correcting errors and defects

discovered during system testing is $7,136 per error. Cigital Inc. [Cig07] considers a large application that has 200 errors introduced during coding.

According to industry average data, the cost of finding and correcting defects during the coding phase is $977 per defect. Thus, the total cost for correcting the 200 "critical" defects during this phase (200 $977) is approximately $195,400.

Industry average data shows that the cost of finding and correcting defects during the system testing phase is $7,136 per defect. In this case, assuming that the system testing phase revealed approximately 50 critical defects (or only 25% of those found by Cigital in the coding phase), the cost of finding and fixing those defects (50 $7,136) would have been approximately $356,800. This would also have resulted in 150 critical errors going undetected and uncorrected. The cost of finding and fixing these remaining 150 defects in the maintenance phase (150 $14,102) would have been $2,115,300. Thus, the total cost of finding and fixing the 200 defects after the coding phase would have been $2,472,100 ($2,115,300 $356,800).

Even if your software organization has costs that are half of the industry average (most have no idea what their costs are!), the cost savings associated with early quality control and assurance activities (conducted during requirements analysis and design) are compelling.

POSSIBLE QUESTIONS

PART – B

1. Describe in detail about black box testing.
2. Elucidate basis path testing and write the steps to derive the test cases.
3. List and explain different types of testing done during the testing phase.
4. Discuss about software testing fundamentals to find the most errors with a minimum of effort.
5. Explain the white box testing in detail.
6. Discuss about control structure test case design with example.
7. Explain the software quality concepts in details.
8. How the quality of the software is ensured? Explain.
9. How to perform the quality control and assurance activity in software project?
10. Write minimum 5 test cases to validate user Login Screen.

KARPAGAM ACADEMY OF HIGHER EDUCATION					
CLASS: III BSC CS COURSE NAME: SOFTWARE ENGINEERING COURSE CODE: 15CSU601 UNIT: V BATCH-2015-2018					
ONE MARKS					
Questions	opt1	opt2	opt3	opt4	Answer
Validation focuses on _____.	the ability of the interface to implement every user task correctly	the degree to which the interface is easy to use and easy to learn.	the user's acceptance of the interface as a useful tool in their work.	all of the above	all of the above
_____ is a critical element of software quality assurance and represents the ultimate review of specification, design, and code generation.	software specification	software generation	software coding	software testing	software testing
Software is tested from _____ different perspectives.	2	3	4	5	2
Software engineers are by their nature _____ people.	pessimistic	optimistic	constructive	destructive	constructive
_____ is a process of executing a program with the intent of finding an error.	coding	testing	debugging	designing	testing
All tests should be _____ to customer requirements.	traceable	designed	tested	coded	traceable
Tests should be planned long before _____ begins.	testing	coding	specification	requirements	testing
Testing should begin in the _____ and progress toward testing in the large.	design	beginning	small	big	small
The less there is to test, the more _____ we can test it.	quickly	shortly	automatically	hardly	quickly
_____ is a process of executing a program with the intend of finding an error.	testing	coding	planning	designing	testing
A good _____ is one that has a high probability of finding an as-yet-undiscovered error	planning	test case	objective	goal	test case
All _____ should be traceable to customer-requirements.	analysis	designs	tests	plans	tests
_____ is simple how easily a computer program can be tested.	software operability	software simplicity	software decomposability	software testability	software testability

The better it works, the more efficiently it can be testing. This characteristic is called _____.	operability	observability	controllability	decomposability	operability
There are _____ characteristics in testability	5	6	7	8	7
What you see is what you test. This characteristic is called _____.	controllability	observability	decomposability	stability	observability
The better we can control the software, the more the testing can be automated and optimized. This characteristic is called _____.	operability	stability	understandability	controllability	controllability
By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting. This characteristic is called _____.	decomposability	simplicity	stability	understandability	decomposability
The less there is to test, the more quickly we can test it. This characteristic is called _____.	controllability	simplicity	operability	observability	simplicity
The fewer the changes, the fewer the disruptions to testing. This characteristic is called _____.	controllability	decomposability	stability	understandability	stability
The more information we have, the smarter we will test. This characteristic is called _____.	controllability	decomposability	stability	understandability	understandability
A good test has a high _____ of finding an error.	probability	simplicity	understandability	stability	probability
A good test is not _____.	stable	redundant	simple	complex	redundant
White-box testing sometimes called _____.	control structure testing	condition testing	glass-box testing	black-box testing	glass-box testing
Logic errors and incorrect assumptions are inversely proportional to the _____ that a program path will be executed	simplicity	probability	understandability	stability	probability
Typographical errors are _____.	redundant	simple	random	complex	random
One often believes that a _____ path is not likely to be executed when, in fact, it may be executed on a regular basis.	control	structural	physical	logical	logical
Basic path testing is a _____.	black-box testing	white-box testing	control structure testing	control path testing	white-box testing

_____ is a software metric that provides a quantitative measure of the logical complexity of a program.	cyclomatic complexity	flow graph	deriving test cases	graph matrices	cyclomatic complexity
An _____ is any path through the program that introduces atleast one new set of processing statements or a new condition.	dependent path	independent path	basic path	control path	independent path
There are _____ steps to be applied to derive the basis set.	2	3	4	5	4
There are _____ test cases that satisfy the basis set.	3	4	5	6	6
. A _____ is a square matrix whose size is equal to the number of nodes on the flow graph.	graph matrix	matrix	flow graph	cyclomatic complexity	graph matrix
To develop a software tool that assists in basis path testing, a data structure called a _____ is useful.	matrix	flow graph	graph matrix	cyclomatic omplexity	graph matrix
_____ requires three or four tests to be derived for a relational expression.	branch testing	data flow testing	data control testing	domain testing	domain testing
_____ is probably the simplest condition testing strategy.	branch testing	data flow testing	condition testing	domain testing	branch testing
The _____ method selects test paths of a program according to the locations of definitions and uses of variables in the program	data flow testing	condition testing	loop testing	black box testing	data flow testing
_____ is a white box testing technique that focuses exclusively on the validity of loop constructions	data flow testing	loop testing	condition testing	control path testing	loop testing
_____ is a test case design method that exercises the logical conditions contained in a program module	black box testing	loop testing	data flow testing	condition testing	condition testing
_____ is called behavioral testing.	black box testing	loop testing	data flow testing	condition testing	black box testing
The first step in _____ is to understand the objects that are modeled in software and the relationships that connect these objects	black box testing	loop testing	data flow testing	condition testing	black box testing

Equivalence partitioning is a _____ method that divides the input domain of a program into classes of data.	black box testing	loop testing	data flow testing	condition testing	black box testing
Comparison testing is also called _____.	black box testing	loop testing	behavioral testing	back-to-back testing	back-to-back testing
_____ testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing.	orthogonal array	loop	behavioral	back-to-back	orthogonal array
_____ focuses verification effort on the smallest unit of software design – the software component or module.	module testing	unit testing	structure testing	system testing	unit testing
A driver is nothing more than a _____.	subprogram	main program	stub	subroutine	main program
_____ serve to replace modules that are subordinate called by the component to be tested.	subprograms	main programs	stubs	subroutines	stubs
Drivers and _____ represent overhead.	subprograms	main programs	stubs	subroutines	stubs
_____ of execution paths is an essential task during the unit test.	unit testing	module testing	selective testing	white box testing	selective testing
Good _____ dictates that error conditions be anticipated and error-handling paths set up to reroute or cleanly terminate processing when an error does occur	design	testing	code	module	design
_____ is completely assembled as a package, interfacing errors have been uncovered and corrected.	software	program	code	all of the above	software
All tests should be _____ to customer requirements.	traceable	designed	tested	coded	traceable

Reg. No.....

[12CSU601]

KARPAGAM UNIVERSITY

(Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021
(For the candidates admitted from 2012 onwards)

B.Sc., DEGREE EXAMINATION, APRIL 2015

Sixth Semester

COMPUTER SCIENCE

SOFTWARE ENGINEERING

Time: 3 hours Maximum : 100 marks

PART – A (15 x 2 = 30 Marks)
Answer ALL the Questions

1. Define Software Engineering.
2. Define spiral model,
3. Mention the role of software.
4. Define Prototyping.
5. Mention the advantages in Simple Substitution ciphers.
6. Define Data Modeling.
7. Write about process specification.
8. What is Behavioral model?
9. Write about Design process.
10. Define Software Architecture.
11. Define Data Flow.
12. Define user interface analysis.
13. Write about Data integrity techniques.
14. What is called as process?
15. Define user analysis.

PART B (5 X 14= 70 Marks)
Answer ALL the Questions

16. a. Explain the role of software-myths in detail.

Or

b. Explain the waterfall model and incremental process model.

17. a. Give an account of data-object-data attributes-relationship and cardinality.

Or

b. Write about the control specification in detail.

18. a. Explain the Process and Design quality.

Or

b. Write about Task analysis and modeling.

19. a. Explain user interface analysis and design.

Or

b. Explain user interface design patterns.

20. **Compulsory :-**

Explain about Quality concepts.

Reg. No.....

[13CSU601]

KARPAGAM UNIVERSITY

Karpagam Academy of Higher Education
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021
(For the candidates admitted from 2013 onwards)

B.Sc., DEGREE EXAMINATION, APRIL 2016

Sixth Semester

COMPUTER SCIENCE

SOFTWARE ENGINEERING

Time: 3 hours Maximum : 60 marks

PART – A (20 x 1 = 20 Marks) (30 Minutes)
(Question Nos. 1 to 20 Online Examinations)

PART B (5 x 8 = 40 Marks) (2 ½ Hours)
Answer ALL the Questions

21. a. Explain the layered technology of software engineering.

(Or)

b. Write about the waterfall model.

22. a. Write a short note on elements of the analysis model.

(Or)

b. Write a note on creating a control flow model.

23. a. Explain about the Design Process.

(Or)

b. Describe the Software Architecture.

24. a. Give an detailed notes on Interface design.

(Or)

b. Explain about the Design evaluation.

25. a. Explain about the detailed concept of white-box testing.

(Or)

b. Write a note on data-flow testing.

Reg. No.....

[14CSU601]

KARPAGAM UNIVERSITY

Karpagam Academy of Higher Education
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021
(For the candidates admitted from 2014onwards)

B.Sc., DEGREE EXAMINATION, APRIL 2017

Sixth Semester

COMPUTER SCIENCE

SOFTWARE ENGINEERING

Time: 3 hours

Maximum : 60 marks

PART – A (20 x 1 = 20 Marks) (30 Minutes)
(Question Nos. 1 to 20 Online Examinations)

PART B (5 x 8 = 40 Marks) (2 ½ Hours)
Answer ALL the Questions

21. a. Describe the concept of layered technology.

Or

b. Briefly explain about the spiral model.

22. a. Explain about the data modeling concepts.

Or

b. How to create a control flow model? Explain.

23. a. Describe the techniques of design concepts.

Or

b. Write a short note on architectural design.

24. a. Explain about user interface analysis and design.

Or

b. Write a note on design evolution.

25. a. Write about the control structure testing.

Or

b. Give a note on black-box testing.

Reg. No.....

[11CSU601]

KARPAGAM UNIVERSITY
(Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021
(For the candidates admitted from 2011 onwards)

B.Sc. DEGREE EXAMINATION, APRIL 2014
Sixth Semester

COMPUTER SCIENCE

SOFTWARE ENGINEERING

Time: 3 hours Maximum : 100 marks

PART – A (15 x 2 = 30 Marks)
Answer ALL the Questions

1. Define the term 'Software'?
2. What do you mean 'Software Myths'?
3. Define the term Prototyping?
4. Explain the term Requirement Analysis?
5. Given the definition of Data attributes?
6. What is the use of Flow Oriented Modeling?
7. Define the term Software design?
8. Explain about Data design?
9. Define the term Transform mapping?
10. Given the names of Golden Rules?
11. List the Stepping design Evaluation?
12. Define the term Workflow analysis?
13. Explain the term Testability?
14. Define the term Condition Testing?
15. What you meant by White-box Testing?

PART B (5 X 14= 70 Marks)
Answer ALL the Questions

16. a) Explain various concepts in the Layered Technology?
Or
b) Discuss briefly about Spiral Model in detail?

17. a) What are the various factors in Requirement Analysis?
Or
b) How to create a Behavioral model?

18. a) Briefly explain the various Design concepts?
Or
b) Discuss various factors in Architectural design?

19. a) List the possible steps in Interface analysis?
Or
b) Explain various design issues in detail?

20. a) What are the fundamental concepts in Software Testing?
Or
b) Discuss various factors in Black-box Testing?
