[MAZZA 94] Mazza and Nelson, “Software Engineering Standards”, Prentice-Hall, 1994.

[PRES93] R. S. Pressman “Software Engineering, A Practitioner’s Approach,” McGraw-Hill, 1993.

[QED89] “CASE the Potential and the Pitfalls,” QED Information Sciences, Inc., QED Plaza, P.O. Box 181, Wellesley, Massachusetts, 1989.

[SEMICH92] Semich, J.W., “Open CASE Emerges as AD/Cycle Lags”, Datamation, March 1992, pp 30-38.

[SMITH90] Smith Connie U. “Performance Engineering of Software Systems”, Addison-Wesley Publishing Co., 1990.

[SOMMERVILLE 92] Sommerville I., “Software Engineering”, Fourth Edition, Addison Wesley, 1992.

[STD94] Software Development and Documentation Standard, MIL-STD-498, US Department of Defence, Washington DC, December, 1994.

[TICE89] Tice G., “software Standards, Proposed standards eases tool interconnection”, IEEE Software, November 1989, pp 69-86.

[YEH93] Hsiang-Tao Yeh “Software Process Quality,” McGraw-Hill Systems Design and Implementation Series, 1993.

2.10 REFERENCES

[AYER92] S. J. Ayer, and F. S. Patrinostro “Software Configuration Management,” McGraw-Hill Systems Design and Implementation series, 1992.

[BOEHM 88] B. W. Boehm “A Spiral Model for Software Development and Enhancement,” IEEE Computer 21, no. 5, pp 61-72, May 1988.

[BROWN92] Brown, Alan., Mcdermid, John., “Learning from IPSE’s Mistakes”, IEEE Software, March 1992, pp 23-28.

[DORFMAN&THAYER 90] Standards, guidelines, and examples on system and software requirements engineering, M. Dorfman and R. Thayer, IEEE Comp. Soc. press tutorial, 1990.

[DOWN 94] Alex Down, Michael Coleman, and Peter Absolon Risk Management For Software Projects, The IBM McGraw-Hill Series, McGraw-Hill, 1994. ISBN 0-07-707816-0

[FERN 92] Fernström C., Närfelt K., Ohlsson L., “Software Factory Principles, Architecture, and Experiments”, IEEE Software, March 1992, pp 36-44.

[GOMA84] Gomma, H., “A Software Design Method for Real-Time Systems,” CACM, vol. 27, no. 9, 1984, pp 938-949.

[HUM89] Watts S. Humphery “Managing the Software Process,” Addison-Wesley, 1989.

[INCE93] Darrel Ince, Helen Sharp, and Mark Woodman “Introduction to Software Project Management and Quality Assurance,” The McGraw-Hill International Series in Software Engineering, 1993.

[LEWIS 91] Lewis T.G., “CASE: Computer-Aided Software Engineering”, Van Nostrand Reinhold, New York, 1991.

11. Describe briefly the purpose and major activities of the software development management process, in general, and the planning activities in particular. Use a single brief sentence for each activity.

12. What is meant by Configuration Management. Discuss the term and importance of configuration identification and configuration change control.

13. Draw a diagram showing the phases and milestones of the ESA standard for software development.

14. Describe briefly the differences between the old DOD-STD-2167A standard for software development and the recent MIL-STD-498 standard.

15. List the various documents and reviews required by the MIL-STD-498 standard.

2.9 EXERCISES

1. Describe the meaning of the term “software life cycle model” in one sentence.

2. Referring to Figure 2.1, discuss how the system concept is established and why capturing the precise user requirements is difficult?, and how is it crucial to the development process?.

3. What is the development process model suggested when the problem of imprecise user requirements exist, as discussed in exercise 2 above. Discuss briefly the properties of this development process model.

4. Figure 2.2 describes the waterfall development model with iterations within each phase.

Summarize in one sentence the function of each phase and why iterations are needed.

5. Discuss the main problems with the waterfall model.

6. Discuss the definition of prototyping in the software engineering terminology, and its effect on the software development process model.

7. Discuss the major features of the incremental model and the evolutionary model of the software development process

8. Table 2.1 in section 2.2.1 gives a typical scenario found in many real-time system development projects. The scenario concludes that the evolutionary model should be used. Change the scenario given in the table by changing the level or deleting or adding risk items or opportunity items in order to conclude that the incremental model is the choice.

9. Repeat exercise 8 above to conclude that the waterfall model is the model of choice.

10. The spiral model shown in Figure 2.7 includes a customer evaluation step inside the development loop. Discuss why this is important in reducing the project risk factor.

physical collection of processors and devices that serve as the platform for execution of the system.

StP/Booch automatically generates class interface and definitions files from the object models. Various environments supported are ANSI 3.0, ARM, and cfront C++.

2.8.7 Document Preparation System (DPS) in StP

The DPS is a report generator. The Document-Templates are used to facilitate the document generation process. The information related to various objects (DDEs, Structured Analysis Diagrams, Structure Charts etc.) is extracted from the project data-base and placed in documents. StP supports document generation in various formats including post-script, Troff, ASCII, Interleaf and FrameMaker.

2.8.6 OOA/OOD by Grady Booch in StP

StP/Booch provides Booch method support for analysis, design and implementation. Other features like checking, documentation, code generation and reverse engineering for reuse are also provided. Various editors used in StP/Booch are summarized below:

- Requirement Table Editor: It is used for high level requirements or business rules. Use-Case diagrams can be created to understand usage models. Event trace diagrams are utilized to describe scenarios for each use case, identify classes and the events passed between them.
- Class Editor: It is used to express classes, their attributes and operations and the relationships between classes.
- Class Table Editor: It is used for editing text version of the graphical models.
- State Transition Editor: it is used to model dynamic behavior. This helps to understand the operations that must be performed by the class in response to events identified in the use case editor.
- Object Interaction Editor: This is used to model how objects will interact. This helps analyze the events which will invoke the class operations and how interactions are sequenced. These models are used during detailed design.
- Module Diagram Editor: It is used to show the allocation of classes and objects to modules in the physical design of a system to gain better understanding of the physical layering and partitioning of the architecture.
- Process Diagram Editor: This is used to show allocation of processes to processors in the physical design of a system. Process diagrams are useful for determining the

The development process support is available in the form of CRUD Table Editor, data administration control. Various reports such as Management Reports and Normalization and Repository Reports can be generated by using this tool. SQL generation support for RDBMS includes DDL (Data Definition Language), DML (Data Manipulation Language) and DCL (Data Control Language) is provided as well.

2.8.5 Object Modeling Technique (OMT) in StP

StP/OMT provides various editors to support Object Modeling Technique. These are listed below:

- Think Editor: It is used for high level analysis of classes and their interactions. Use-Case diagrams can be created to understand usage models. Event trace diagrams are provided to identify classes and the events that pass between them.
- Object Model Editor: It is used to express classes, their attributes and operations and the relationships between classes.
- Dynamic Model Editor: it is used to model dynamic behavior. Analysis of sequence of events for class operations and timing constraints can be done.
- Functional Model Editor: It is used to develop data flow diagrams which show the operations (data and data transformation) associated with a class.
- Object Interaction Diagrams: These are used to model how objects will interact. Analyst can look at the events which will invoke the class operations and how interactions are sequenced. These models are used during detailed design.
- Class Table Editor: it is used for editing text version of the graphical models.

StP/OMT automatically generates class interface and definitions files from the object models. Various environments supported are ANSI 3.0, ARM, and cfront C++.

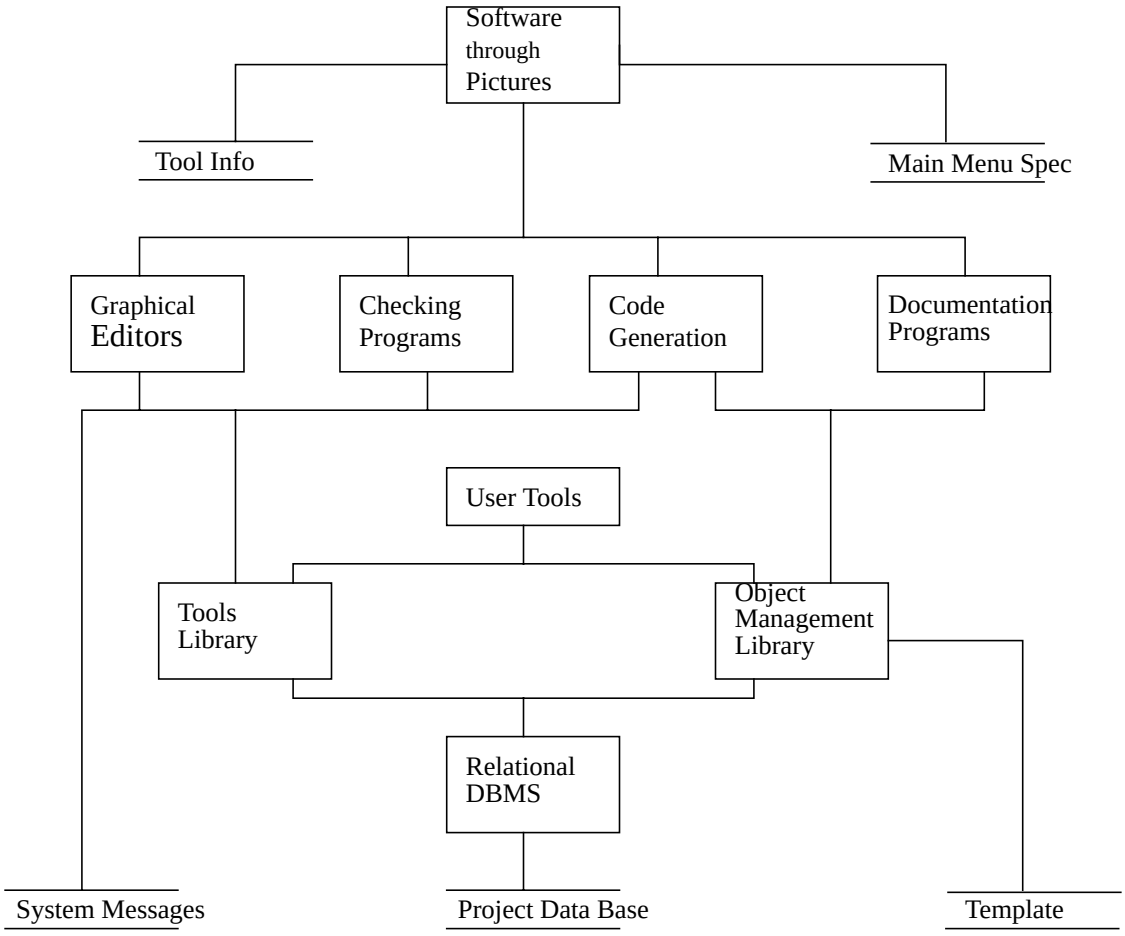


FIGURE 2.15 The architecture of StP

2.8.4 Information Modeling (IM) in StP

StP/IM supports conceptual, logical models and development process. For the conceptual and logical models, the support includes the Entity Relationship Editor for various notations. These notations are by Bachman, Martin and Chen with extended attributes. It allows alternative attribute representations and automatic key inheritance. Also available are the Data Element Table Editor and Domain Table Editor. It provides capabilities to extract a logical model from a conceptual model as well.

2.8.2 Real-time System's Modeling in StP

Hatley/Pirbhai notation is used to model the event-driven aspects of real-time systems. The editor provides standard symbols like control flows, control stores, control specifications etc. These symbols are used in conjunction with structured analysis symbols to complete the Structured Analysis Diagrams of real-time systems. Control specification are defined using Control Specification Editor and State Transition Diagram Editor. The integrated checking programs are used to check:

- the individual control-specification (c-specs),
- unique initial states and events in c-specs tables,
- balancing of control flow in a c-spec with the associated control flow diagram,
- DDE consistency between state transition diagrams and c-specs,
- DDE consistency among c-spec tables,
- DDE control -in-flows that are required to be generated in c-spec tables.

2.8.3 Structured Design in StP

For the structured design in StP, Constantine/Yourdon notation is used. The symbols provided are User-defined modules, library modules, lexical inclusion modules, global data modules, iteration, conditional, on-page connectors, parameters (in, out, in-out directions) and flags (in, out). As in case of Structured Analysis, the Structured Charts can be decomposed to form an hierarchical Design model.

PDL (Program Design Language) can be generated for any one or all Structure Chart Modules. Specific programming language code skeletons can be substituted in lieu of PDLs to generate source code. The files outputs for PDL code skeletons can be edited by the user. Structured Design in StP is supported by other tools such as PDL checkers and compilers.

2.8 ICASE Tool: Software through Pictures (StP) by IDE

The tool-suite of Software through Pictures is divided into four sets of tools as shown in the Figure 2.15. The detail of these sets is given below:

- Graphical Editors
- Checking Programs
- Code Generation
- Documentation programs

The data related to all of these tool-sets is stored in the project data-base. StP supports various notations for analysis and design of software-systems. A brief overview of this is given below:

2.8.1 Structured Analysis in StP

The structured analysis in StP is based on the notations like DeMarco/Yourdon, Bane/Sarson. The standard symbols available are externals, processes, data flows, data stores, and off-page connectors. The functional aspects of the software-systems is captured using the Datival-Diagram-Editor. The editor has capabilities to allow the decomposition of diagrams for a hierarchical representation of the software-system. Thus a process is decomposed until it becomes a “primitive process”. A primitive process can not be decomposed any further. At this stage StP provides the capabilities to generate process specification from a user-definable template.

The integrated checking programs are used to look at the Data-Dictionary, Decompositions and the Structured Analysis Diagrams. This includes balancing of data flows, processes, and data stores. Data Dictionary entries are checked for consistency.

- A common database minimizes control coupling between tool. Because tools operate on the state of the database, rather than the direct output of another tool, they can be ignorant of when the data was produced and who produced it.
- Despite the advantages (listed above), database-centered environments are not widely used . There are a couple of reasons behind it, as listed below:
    * The data models of conventional database-management systems can not express the rich semantics that CASE applications require (e.g. complex integrity constraints and derivation dependencies among entities). If the schema can not explicitly state such knowledge, it must be stated in the code. And if several tools share the same data, this code must be included in all of them. Such duplication of code reduces tool independence and makes it harder to integrate them.
    * Different tools have very different requirements on issues such as transaction model, query model, aggregation facilities and information granularity. It is difficult to construct a DBMS that combines a data model of sufficient semantic richness, the generality to cover varying requirements and the high performance that interactive environments require.

There is an effort by some tool-suite developers to go around these limitations by building language-centered environments. But they achieve tight-integration by building on special purpose databases instead of commercial DBMSs. These environments neither fit into a wider context nor cooperate easily with other tools.

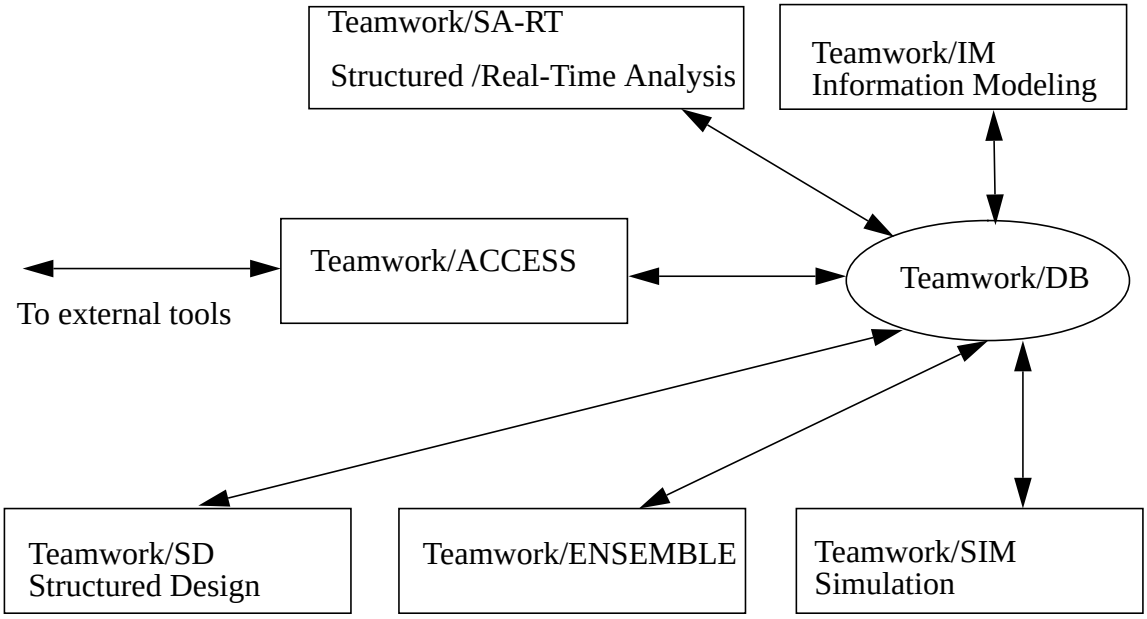


FIGURE 2.14 Example of Tightly-Coupled Data Integration Architecture

A common database facilitates the tool-integration in many ways as listed below:

- Its data model imposes a uniform format for all data that is used by more than one tool.
- Its shared data enables the short response times that highly interactive environments require.
- A common database schema explicitly expresses the database's intention and how it relates pieces of information. This expression is distinct from how the data is actually stored. Experience has shown that the use of schemas forces developers to give precise information descriptions, thus reducing misinterpretation. Separating semantic data descriptions from storage representation also results in systems that are easy to make small changes to.

interface of classes, and internal workings of classes. C++ code can be generated from the diagrams.

Alongside the tools listed in the preceding paragraphs, teamwork has documentation tools as well. A brief description of these tools is given below:

2.7.1.9 DPI (Document Production Interface)

This is a document layout tool. It is used to create a template to document models. It supports DOD-STD 2167 and 2167A documentation requirements.

2.7.1.10 Teamwork/DocGen

It produces Interleaf documents. This tool is used to create a template to document models. It supports DoD-STD-2167 and 2167A documentation requirements. These are the standards existed prior to the MIL-STD-498 standard.

2.7.1.11 DocConnect

This is a set of programs and scripts that work with Teamwork and Interleaf to ease the process of retrieving and maintaining Teamwork objects in the Interleaf documents. It allows the user to create links from the Interleaf component bar, copy objects from Teamwork indexes and paste them into documents.

2.7.2 Data Integration Architecture of Teamwork

Most of the recent architectures for software development environments are based on the use of a central database, or repository, which stores all relevant data . An example of tightly-coupled data integration architecture is shown in Figure 2.14. All the tools share the same database and use the same standards for transferring the information between each other.

2.7.1.5 Teamwork/TestCase

Teamwork/TestCase is used to generate test cases in analysis or design stage. Once the DFD's or Structure Charts are ready, the DDE's (Data Dictionary Entries) are used to setup the model. This model is then used by TestCase to generate test cases. The test cases generated in the analysis phase can be used by Teamwork/DA to determine possible outcomes of various testing scenarios.

2.7.1.6 Teamwork/Ada

This tool uses Buhr (with extensions) standard. It helps systems engineers create, Ada system designs and code. A cross-reference capability is provided to help locate object dependencies in a given model. Ada Structure Graph (ASG) Editor lets designers build, and modify Ada design elements. Ada Source Builder (ASB) analyzes design diagrams and produces compilable Ada code frames for specifications and also for program unit bodies. The ASB checks diagrams against established criteria to verify the integrity of the design. Design Sensitive Editor (DSE) helps to create Ada source code. Teamwork/DSE is sensitive to Ada design and to the Ada language syntax.

2.7.1.7 Teamwork/OOA

Using Shlaer/Mellor standard, this tool facilitates Object-Oriented Analysis. Teamwork/OOA helps analyze problems in terms of information, state, and process models.

2.7.1.8 Teamwork/OOD

Teamwork/OOD uses Project Technology notation (Object-Oriented Design Language or OODLE) in Systems Design phase. It allows the designer to create Object-Oriented Design Diagrams. It uses four diagrams to show dependencies, inheritance, the public

edit, software specifications consisting of Data Flow Diagrams, Process Specifications, and Data Dictionary Entries stored in a common project database.

Hatley/Pirbhai notation is the base of Teamwork/RT. This tool is used in conjunction with teamwork/SA. It helps analysts and designers create, store, review, and maintain structured real-time software specifications. With Teamwork/RT, users can create and edit software specifications for real time software-systems that deal with issues of sequence, timing, and control. The software-system specifications consist of Data and Control Flow Diagrams, Control Specifications, Process Specifications, and Data Dictionary Entries stored in a common project database.

2.7.1.3 Teamwork/DA

Teamwork/DA supports system analysis and design by enabling the simulation of a Teamwork/SART model. Teamwork/DA enables the execution and testing of the model. It supports the ability to simulate the behavior and performance of a hardware and software system under a variety of input and performance conditions. It also allows the development of alternative models and compares their performance. A detailed view of this tool is given in Chapter 6.

2.7.1.4 Teamwork/SD

This tools is based on Yourden and Constantine’s method that helps analysts and designers create, structured software designs. It supports a rigorous definition process and ensures product quality by checking design specifications for cohesion and coupling and ensuring quality of design, accuracy, completeness, syntax, and correct balancing. With Teamwork/SD, users can create and edit software designs, which consist of Structure Charts, Module Specifications, and Data Dictionary Entries.

2.7 ICASE Tool: Teamwork by Cayenne Software Inc.

The ICASE environments discussed here is, teamwork developed by Cayenne Software Inc. (former Cadre Technologies Inc.). Various phases covered by this suite of tools is given below:

- Analysis Phase: teamwork/IM, teamwork/SA-RT, teamwork/SIM, teamwork/OOA
- Design Phase: teamwork/SD, teamwork/ADA, teamwork/OOD
- Test-case generation: teamwork/Test Case
- Coding and Code-Generation Phase: teamwork/Ensemble, Ada Source Builder
- Test verification: teamwork/Ensemble

The teamwork tool suite is discussed in the following sections.

2.7.1 Teamwork tool suite

2.7.1.1 Teamwork/IM

Following Chen's methodology, this tools helps analysts create, store, review, and maintain complex data models. It promotes a rigorous definition process and ensures quality by checking models for accuracy, completeness, syntax, and correct balancing. Users of Teamwork/IM can create and edit graphic, verifiable data models, which consist of Entity Relationship Diagrams and Data Dictionary Entries stored in a common project database.

2.7.1.2 Teamwork/SA-RT

Teamwork/SA is a tool based on Yourden/DeMarco. It helps analysts and designers to create, store, review, and maintain structured software specifications. It facilitates rigorous definition process and ensures quality by checking specifications for accuracy, completeness, syntax, and correct balancing. With Teamwork/SA, users can create and

CDIF is also a model of how systems should be built [SEMICH92]. The basis of CDIF is agreed-on syntactic definitions that let tools exchange the data [BROWN92].

2.6.2.3 Information Resource Dictionary System (IRDS)

An example of syntactic level integration standard is the IRDS. This is an Entity-relationship based model that describes the way information is logically stored in the repositories and the methods to be used by tools to access the information. It is a data-base schema which is good for tightly-coupled integration architectures. IRDS defines what should be contained in a repository but leaves its technical design and implementation methods undefined.

2.6.2.4 Portable Common Tool Environment (PCTE)

PCTE provides a broad and complex set of interface calls (similar to operating-system calls) to underlying facilities that support CASE tools. But it has no explicit functions to support software engineering [BROWN92]. It permits the use of CASE tools across operating systems and LANs. PCTE was developed under the European Strategic Program for Research in Information Technology (ESPRIT). It is supported by several European vendors. It includes a common object oriented layer for transparent information exchange between different types of tools [SEMICH92]. It is primarily used for C, C++ and Ada environments.

standards are being used at present time. These standards are described in the following section.

2.6.2 The Standards for Tool Integration.

Several standards were proposed and implemented to exchange data between ICASE environments developed by different vendors. These standards are briefly described as follows.

2.6.2.1 Semantic Transfer Language (STL)

The IEEE tool integration standard 1175 proposes a Semantic Transfer Language (STL). This is for non-graphical communications among CASE tools. STL can represent a set of CASE tool information with text and generalized graphical information. It attempts to add more semantics in the information to be exchanged by different tools. It is used as an intermediary language to express a powerful set of tool-generated information among different tools in a conceptualized non-graphical form. A detailed view of this standard is given in [TICE89].

2.6.2.2 Case Data Interchange Format (CDIF)

CDIF is defined by the Electronic Industries Association as a standard for CASE tool and repository communication. The salient features of this standard are listed below:

- CDIF is used for exchanging information among CASE tools and repositories.
- It includes descriptions, placement and details of text and graphical elements.
- CDIF based syntactic/semantics level integration is supported by most CASE tools vendors.

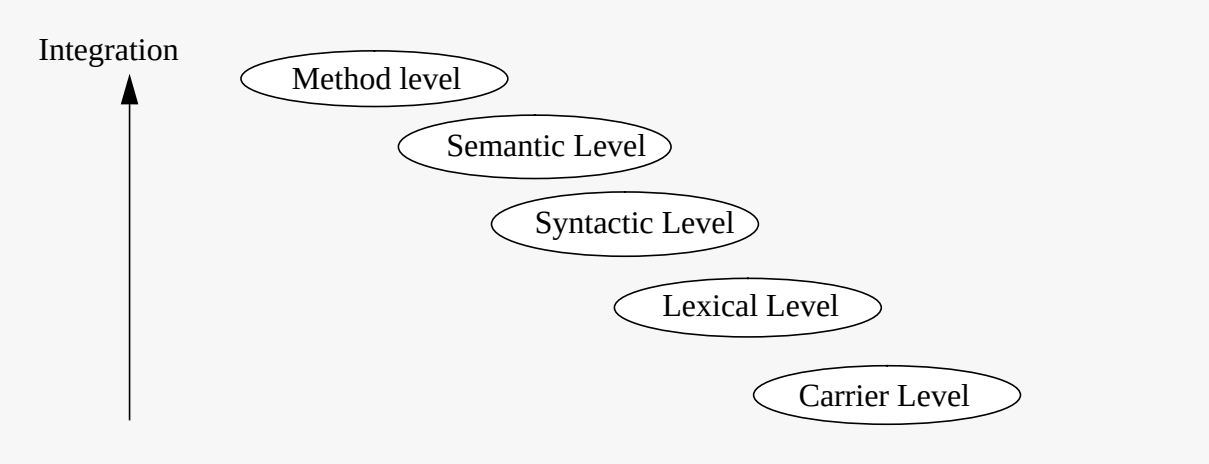


FIGURE 2.13 Levels of tool integration from low (Carrier Level) to high (Method Level)

In semantic level integration, tools agree on the data-structure definitions, as well as the meanings of operations on those structures. At this level there is enough information available to automate development, analysis and design tasks (e.g. code generation tools). This level of agreement between the tools can be achieved by the following methods.

- Hard coding the definition of the data structure and operation specifications into the tools

OR

- Including information about the data structures and operations that make up the infrastructure's data repository in the repository itself. Tools can then query such meta-data.

A common definition of the structures' semantics, augments the syntactic level information. Most of the integrated tool suites use the first approach. As the specifications are defined before the tools are written, tool writers know what structures are available, how they are named, the meaning of each structure, and the effect of each operation. Different integration

2.6 The ICASE Environments

In this section we discuss the integration aspects of Integrated CASE environments. These environments support a variety of development techniques and notations. All the tools used in the development process are presented through a common user interface. Data and diagrams developed in a tool during a particular phase in the development process can be used by another tool in a different phase of development. The layers of tool integration describe or specify the level of tool integration from the weakest form to the strongest form depending on the sophistication of the integration mechanism supported. In the following section integration levels among development and verification tools are described.

2.6.1 Layers of Tool Integration

Integration of tools can be designed at five levels as explained in [BROWN92]. The levels are described below.

- Carrier Level: Tool integration is accomplished by passing byte streams.
- Lexical Level: At this level tools share data formats and operating conventions that make them interact meaningfully.
- Syntactic Level: On syntactic level the tools agree on a set of data structures and on the rules governing their formation.
- Semantic Level: The tools agree on the structure’s semantics which provide enough information to automate development and analysis tasks.
- Method Level: At this level the tools agree on specific process step (e.g. steps of development process such as prototyping, requirements analysis and dynamic modeling).

diagram, dialog flow diagram, and a procedure-action diagram. A code generation tool is also available. The engineering methodology in IEF supports the following steps of information engineering development:

- information strategy planning,
- business area analysis,
- business system design,
- technical system design, construction, and production.

The experience of using CASE at The Hartford Insurance Company in developing information processing systems was very successful. Productivity improvement of up to 300 percent was cited by the Vice President and Director of Information Management at this company. The Application was mainly Information Management Services of transaction processing systems utilizing distributed processing. Due to the complexity of developing distributed processing applications, sequential development using the waterfall life cycle model was not adequate. Prototyping and experimentation was used instead during the requirements phase. In fact a more aggressive approach of using prototyping across the entire life cycle was advocated.

Relational tools by Relational systems Corp. were used for prototyping at The Hartford Insurance Company. A prototype model was generated from an entity structure model, which is based on an event precedence model. An event precedence model in turn is developed based on a business function model as well as a semantic data model. The semantic data model is needed for an extended relational analysis. Data modeling and relational analysis are key steps in developing the event precedence model and for prototyping information management systems.

The experience of using CASE at Amoco is also described in [QED89]. The Information Engineering Facility (IEF), a CASE tool set developed by Texas Instruments, was used in Amoco as a Beta Test site since 1986. The information engineering methodology supported by IEF is a rigorous data-driven methodology consisting of a set of interrelated techniques used to develop and maintain software for informations systems. Information engineering expanded in Amoco and the company has continued information engineering projects, and continued to use the IEF tool set.

The IEF tool set consists of tools to support software analysis and design using entity relationship diagrams, data structures diagram, screen designer (for screen layouts), process dependency

CASE environment at The Hartford Insurance Company, and experiences with implementing systems using the Information Engineering facility (IEF) at Amoco are also discussed. These examples, which prove that CASE tools were successfully used by the business community, are briefly described below.

AT&T developed a prototype of a CASE tool in house in 1984, since the CASE tool market at that time had nothing much to offer. Users of the prototype estimated an 10 to 50 percent productivity gains with even higher estimates for quality gains. Due to the high cost of developing such a tool, a decision was made a few years later to do an industry search for CASE tools. A set of detailed requirements for CASE tool selection were developed. CASE tools were evaluated based on what they can provide. The Bachman Information Systems tools (which were not developed in its entirety) were selected for long term strategic directions, and the Excelerator tool was selected for the short term use. The Bachman tool is designed for both forward engineering and reverse engineering, and the methodology includes expert systems, MS-DOS workstation and high resolution graphics.

The well known Excelerator tool, developed by Index technology Corp., Cambridge, Massachusetts, is quite popular among the PC platform users. It supports the early phases of the development life cycle from feasibility, requirements analysis, and design. The tool provides graphics facilities to develop data flow diagrams, structure charts, logical data models, entity relationship diagrams, and presentation graphs. The Excelerator dictionary (or database) contains information such as data structures, process models, screen definitions, report layout, and system design diagram. The report generation facility provides reports generated directly from the dictionary for document preparation.

the manufacturing functions in CAM, but also the business, management (e.g., computer-aided process planning), and other engineering functions.

Engineering teams may now cooperate concurrently to model and analyze complicated systems using CAD based concurrent engineering software environments. In general, computer modeling and simulation tools (e.g. using finite element modeling) are being heavily used during the engineering development process in many engineering disciplines.

Computer-Aided Software Engineering (CASE) is relatively new compared to CAE tools in many other engineering branches. CASE tools are becoming the most important technological advance in the history of software development. They already have a major impact on the software engineering process in exactly the same way as earlier CAD/CIM tools did to other engineering disciplines. CASE tools now are being used in many companies throughout the industry and their use is encouraged and specified in software engineering development standards as mentioned in section 3 of this chapter.

The maturing process, techniques, and standards of software development gave the necessary fertile ground for the rapid evolution of CASE tools. Structured analysis and design techniques which have been practiced for many years, their maturing notations supported by most CASE tools, the techniques for requirements traceability, support for dynamic modeling and simulations, code generation, test case generation, and reverse engineering are some of the many examples of CASE tools support throughout the development process. Management and project support tools for project sizing, cost estimation, and risk analysis are also available.

Experiences of several companies with CASE tools were described in a book published by QED information sciences Inc., several years ago [QED89]. The book describes experiences of CASE tool selection and experiences with the Excelerator CASE tool at AT&T. Experiences with a

2.5 Evolution of the ICASE Technology

The roots of today's Engineering development technology go back to the beginning of civilization when the use of graphical models was acknowledged by the engineers of ancient Egypt. Some of the existing drawings on the ancient Egyptian tombs are classified as technical drawings. Indeed, the existing monuments of this civilization, and the complexity of developing such monuments, speaks for the historical roots of the engineering development process.

Coming back to our present time, the engineering development process saw a dramatic change with the evolution of information technology and the introduction of computer-aided design (CAD) tools, Computer-integrated Manufacturing (CIM) tools, and Computer- Aided Engineering (CAE) tools.

Electrical Engineers currently use CAE tools to design their hardware using computer schematic diagrams and then model and verify the behavior of the hardware design using computer simulations. Producing layouts for prototyping, VLSI design, and printed circuit boards are also done using CAE tools.

The core of CAD tools in Mechanical Engineering consists of geometric modeling and graphics applications. The tools facilitate the structuring of geometric models using color, grids, and geometric modifiers. It provides manipulations including transformation of the model in space, visualization using shaded images, and animation which help in design conceptualization, communication, and interference or inconsistency detections.

Computer-Aided Manufacturing (CAM) tools include a large number of functions ranging from Flexible Manufacturing systems scheduling to machine control. CIM, however, includes not only

- Defining concurrent tasks which are becoming more and more necessary as needs for multiple functions grow and with the advent of hardware support for multitasking and multiprocessing.
- Specifying communication and synchronization mechanisms among concurrent tasks. Since tasks may execute simultaneously on multiple processors and share resources. These mechanisms are supported by operating systems primitives, a higher level synchronization handling software developed over the operating system primitives, or by the run time environment of a programming language such as Ada.
- Handling a wide variation of data and communication rates. This problem is becoming even more visible with the growing link between computing, communication, and telecommunication networks. This is in addition to the different types and mix of data in terms of real-time digital video images, digital audio signals and text information.
- Dealing with the dependency of the software design and its real-time performance on the operating system, and the external hardware devices which might be changing with the rapid change of technology in this field.

Several methods and approaches for real time software design have been proposed. One of the earlier and well known design methods was developed by Hassan Gomma [GOMA84]. This method extends a data flow diagram based design to represent task concurrency, communication, and synchronization. This method and others will be discussed in detail in Chapters 4 and 5.

A case study on a real-time embedded software application (a disk controller subsystem) is developed and modeled using simplified techniques outlined in [SMITH90]. Simplified analysis using a spreadsheet are used to predict the software performance. This effort although quite interesting in its presentation, does not attempt to link performance analysis models to the elaborate static analysis models, supported in ICASE tools.

The credibility or correctness of simplified models or models which does not correspond directly to the static analysis models are always in doubt and needs to be verified. This is due to the risk in neglecting important behavioral details of the system which might have an indirect significant impact on the system performance. A typical example of this is the highly variable synchronization and communication delays in many concurrent real- time systems.

The ICASE tool support is becoming essential to minimize the time and effort in developing detailed and credible performance models. In this case static analysis models using structured analysis or object-oriented analysis are directly transformed into simulation models. These tools will be discussed in the next section and an example of dynamic modeling will be presented in Chapter 6.

The above performance models are refined during the design phase, where the details of task concurrency, function allocation to tasks (i.e., the task structure), and synchronization between communicating tasks are identified. The scheduling problem between higher priority tasks and lower priority tasks is also analyzed using these models with the aid of more simplified analytic techniques.

2.4.2 Design Methods for Real-Time Systems

The design of real-time systems as mentioned above, becomes much more difficult for the design engineer when problems like the ones given below, are considered:

These parameters are estimated from the hardware requirements of the system and based on previous experience in similar projects.

The data transfer rate on the other hand, specifies how fast data must be moved in or out of the system. This measure is particularly important when a continuous stream of incoming data must be processed without losing or missing any data, and a stream of outgoing data must be produced and sustained. A simple example of the importance of data transfer rates is exemplified in communication software.

Performance evaluation modeling involves the construction of hierarchical simulation or analytic models for the system as a whole and for different subsystems. These models are defined and analyzed using scenarios, where typical processing sequences of the system are simulated to predict the systems response time. In addition, a mixture of event types are simulated to estimate how many event types can be handled in a given time period. The dynamic model produced is essential in gaining an understanding of the system's intricate behavior.

Simplified analysis techniques e.g. a spreadsheet is perhaps adequate during the project planning effort in which a "ball-park" analysis is required. These techniques are highly inadequate to capture the complex real-time behavior. These models can be used as guidance in constructing the detailed performance models where functions, events, and actions are explicitly modeled.

In her book on Performance Engineering of software systems [SMITH90], Connie Smith presented the first comprehensive and yet practical coverage to the problem of software performance modeling. The book however lacks the connection between the performance models and the software analysis models supported by many ICASE tools.

2.4 Real-Time Software Development

The development of real-time software in many applications is complicated by the requirements of high accuracy, safety, and reliability as well as stringent real-time constraints. Functions may be characterized as being periodic, i.e., activated in specified time intervals, or as sporadic, i.e., activated in response to the occurrence of a certain event. These functional characteristics as well as the fault-tolerance requirements on many critical functions greatly complicate the tasks in the analysis, design, implementations, and testing phases. The discussion in this section is focused on the analysis and design phases. Real-time issues concerning implementation and testing will be discussed in later chapters.

2.4.1 Requirements specifications and analysis for real-time systems

In the requirements specification and analysis phase the external timing constraints should be analyzed in the context of a system performance evaluation model. This model is defined from the behavioral specification of the system. Important performance measures include the response time of critical functions and the data transfer rate.

The response time is defined as the time within which an event is detected and an action required by a particular function is executed. Important parameters affecting the response time of a function are:

- the time needed to produce the action, the fault- tolerance capability in terms of the time needed to check the output of the function,
- the time needed in detecting an event (e.g., interrupt latency), and
- the time taken to save the context of the currently running function and loading the context of the required function.

those indicators as described in the plan. Candidate indicators are given in Appendix F of the referenced standard document [STD94].

- Security and privacy: The developer shall meet the security and privacy requirements specified in the contract. These requirements may affect the software development effort, the resulting software products, or both.
- Subcontractor management: If subcontractors are used, the developer shall include in subcontracts all contractual requirements necessary to ensure that software products are developed in accordance with prime contract requirements.
- Interface with software IV&V agents: The developer shall interface with the software Independent Verification and Validation (IV&V) agent(s) as specified in the contract.
- Coordination with associate developers: The developer shall coordinate with associate developers, working groups, and interface groups as specified in the contract.
- Improvement of project processes: The developer shall periodically assess the processes used on the project to determine their suitability and effectiveness. Based on these assessments, the developer shall identify any necessary and beneficial improvements to the process, shall identify these improvements to the acquirer in the form of proposed updates to the software development plan and, if approved, shall implement the improvements on the project.

b) Resolve issues that could not be resolved at joint technical reviews.

c) Arrive at agreed-upon mitigation strategies for near and long-term risks that could not be resolved at joint technical reviews.

d) Identify and resolve management-level issues and risks not raised at joint technical reviews.

e) Obtain commitments and acquirer approvals needed for timely accomplishment of the project.

Other activities

The developer shall perform the following activities.

- Risk management: The developer shall perform risk management throughout the software development process. The developer shall identify, analyze, and prioritize the areas of the software development project that involve potential technical, cost, or schedule risks; develop strategies for managing those risks; record the risks and strategies in the software development plan; and implement the strategies in accordance with the plan.
- Software management indicators: The developer shall use software management indicators to aid in managing the software development process and communicating its status to the acquirer. The developer shall identify and define a set of software management indicators, including the data to be collected, the methods to be used to interpret and apply the data, and the planned reporting mechanism. The developer shall record this information in the software development plan and shall collect, interpret, apply, and report on

a) Review evolving software products, using as criteria the software product evaluation criteria in Appendix D of reference standards document [STD94].

b) Review and demonstrate proposed technical solutions; provide insight and obtain feedback on the technical effort; surface and resolve technical issues.

c) Review project status; surface near- and long-term risks regarding technical, cost, and schedule issues.

d) Arrive at agreed-upon mitigation strategies for identified risks, within the authority of those present.

e) Identify risks and issues to be raised at joint management reviews.

f) Ensure on-going communication between acquirer and developer technical personnel.

- Joint management reviews: The developer shall plan and take part in joint management reviews at locations and dates proposed by the developer and approved by the acquirer. These reviews shall be attended by persons with authority to make cost and schedule decisions and shall have the following objectives. Examples of such reviews are identified in Appendix E of reference standards document [STD94].

  a) Keep management informed about project status, directions being taken, technical agreements reached, and overall status of evolving software products.

c) Each problem shall be classified by category and priority, using the categories and priorities in Appendix C of the referenced standard document [STD94], or approved alternatives.

d) Analysis shall be performed to detect trends in the problems reported.

Corrective actions shall be evaluated to determine whether problems have been resolved, adverse trends have been reversed, and changes have been correctly implemented without introducing additional problems.

Joint technical and management reviews

The developer shall plan and take part in joint (acquirer/developer) technical and management reviews in accordance with the following requirements.

* Note: If a system or CSCI is developed in multiple builds, the types of joint reviews held and the criteria applied will depend on the objectives of each build. Software products that meet those objectives can be considered satisfactory even though they are missing information designated for development in later builds.

- Joint technical reviews: The developer shall plan and take part in joint technical reviews at locations and dates proposed by the developer and approved by the acquirer. These reviews shall be attended by persons with technical knowledge of the software products to be reviewed. The reviews shall focus on in-process and final software products, rather than materials generated especially for the review. The reviews shall have the following objectives:

and organizational freedom to permit objective software quality assurance evaluations and to initiate and verify corrective actions.

Corrective action

The developer shall perform corrective action in accordance with the following requirements.

- Problem/change reports: The developer shall prepare a problem/change report to describe each problem detected in software products under project-level or higher configuration control and each problem in activities required by the contract or described in the software development plan. The problem/change report shall describe the problem, the corrective action needed, and the actions taken to date. These reports shall serve as input to the corrective action system.
- Corrective action system: The developer shall implement a corrective action system for handling each problem detected in software products under project-level or higher configuration control and each problem in activities required by the contract or described in the software development plan. The system shall meet the following requirements:

  a) Inputs to the system shall consist of problem/change reports.

  b) The system shall be closed-loop, ensuring that all detected problems are promptly reported and entered into the system, action is initiated on them, resolution is achieved, status is tracked, and records of the problems are maintained for the life of the contract.

- Software quality assurance evaluations: The developer shall conduct on-going evaluations of software development activities and the resulting software products to:

  a) Assure that each activity required by the contract or described in the software development plan is being performed in accordance with the contract and with the software development plan.

  b) Assure that each software product required by this standard or by other contract provisions exists and has undergone software product evaluations, testing, and corrective action as required by this standard and by other contract provisions.

- Software quality assurance records: The developer shall prepare and maintain records of each software quality assurance activity. These records shall be maintained for the life of the contract. Problems in software products under project-level or higher configuration control and problems in activities required by the contract or described in the software development plan shall be handled as described in 5.17 of the standard referenced document [STD94] (please see the section *Corrective action* in the following pages).

- Independence in software quality assurance: The persons responsible for conducting software quality assurance evaluations shall not be the persons who developed the software product, performed the activity, or are responsible for the software product or activity. This does not preclude such persons from taking part in these evaluations. The persons responsible for assuring compliance with the contract shall have the resources, responsibility, authority,

- Software product evaluation records: The developer shall prepare and maintain records of each software product evaluation. These records shall be maintained for the life of the contract. Problems in software products under project-level or higher configuration control shall be handled as described in 5.17 of reference standard document [STD94] (please see the section *Corrective action* in the following pages).
- Independence in software product evaluation: The persons responsible for evaluating a software product shall not be the persons who developed the product. This does not preclude the persons who developed the software product from taking part in the evaluation (for example, as participants in a walk-through of the product).

Software quality assurance

The developer shall perform software quality assurance in accordance with the following requirements.

* **Note**: If a system or CSCI is developed in multiple builds, the activities and software products of each build should be evaluated in the context of the objectives established for that build. An activity or software product that meets those objectives can be considered satisfactory even though it is missing aspects designated for later builds. Planning for software quality assurance is included in software development planning.

- Configuration audits: The developer shall support acquirer- conducted configuration audits as specified in the contract.
    * **Note**: These configuration audits may be called Functional Configuration Audits and Physical Configuration Audits.
- Packaging, storage, handling, and delivery: The developer shall establish and implement procedures for the packaging, storage, handling, and delivery of deliverable software products. The developer shall maintain master copies of delivered software products for the duration of the contract.

Software product evaluation

The developer shall perform software product evaluation in accordance with the following requirements.

* **Note**: If a system or CSCI is developed in multiple builds, the software products of each build should be evaluated in the context of the objectives established for that build. A software product that meets those objectives can be considered satisfactory even though it is missing information designated for development in later builds.

- In-process and final software product evaluations: The developer shall perform in-process evaluations of the software products generated in carrying out the requirements of this standard. In addition, the developer shall perform a final evaluation of each deliverable software product before its delivery. The software products to be evaluated, criteria to be used, and definitions for those criteria are given in Appendix D of reference standard document [STD94].

units, configuration items. The identification scheme shall include the version/ revision/release status of each entity.

- Configuration control: The developer shall establish and implement procedures designating the levels of control each identified entity must pass through (for example, author control, project-level control, acquirer control); the persons or groups with authority to authorize changes and to make changes at each level (for example, the programmer/analyst, the software lead, the project manager, the acquirer); and the steps to be followed to request authorization for changes, process change requests, track changes, distribute changes, and maintain past versions. Changes that affect an entity already under acquirer control shall be proposed to the acquirer in accordance with contractually established forms and procedures, if any.

    * **Note**: A number of requirements in this standard refer to "project-level or higher configuration control". If "project-level" is not a level of control selected for the project, the software development plan should state how these requirements map to the selected levels.

- Configuration status accounting: The developer shall prepare and maintain records of the configuration status of all entities that have been placed under project-level or higher configuration control. These records shall be maintained for the life of the contract. They shall include, as applicable, the current version/revision/release of each entity, a record of changes to the entity since being placed under project-level or higher configuration control, and the status of problem/change reports affecting the entity.

of the deliverable software after delivery to the acquirer do not depend on the non-deliverable software. Otherwise a provision is made to ensure that the acquirer has or can obtain the same software. The developer shall ensure that all non-deliverable software used on the project performs its intended functions.

Software configuration management.

The developer shall perform software configuration management in accordance with the following requirements.

* **Note**: If a system or CSCI is developed in multiple builds, the software products of each build may be refinements of, or additions to, software products of previous builds. Software configuration management in each build should be understood to take place in the context of the software products and controls in place at the start of the build.

- Configuration identification: The developer shall participate in selecting CSCIs, as performed under system architectural design (see section 5.4.2 of [STD94]). The developer should also identify the entities to be placed under configuration control, and shall assign a project-unique identifier to each CSCI (and each additional entity to be placed under configuration control). These entities shall include the software products to be developed or used under the contract and the elements of the software development environment. The identification scheme shall be at the level at which entities will actually be controlled, for example, computer files, electronic media, documents, software

interpreted to mean establishing the environment needed to complete that build.

- Software engineering environment: The developer shall establish, control, and maintain a software engineering environment to perform the software engineering effort. The developer shall ensure that each element of the environment performs its intended functions.
- Software test environment: The developer shall establish, control, and maintain a software test environment to perform qualification, and possibly other, testing of software. The developer shall ensure that each element of the environment performs its intended functions.
- Software development library: The developer shall establish, control, and maintain a software development library (SDL) to facilitate the orderly development and subsequent support of software. The SDL may be an integral part of the software engineering and test environments. The developer shall maintain the SDL for the duration of the contract.
- Software development files: The developer shall establish, control, and maintain a software development file (SDF) for each software unit or logically related group of software units, for each CSCI, and, as applicable, for logical groups of CSCIs, for subsystems, and for the overall system. The developer shall record information about the development of the software in appropriate SDFs and shall maintain the SDFs for the duration of the contract.
- Non-deliverable software: The developer may use non- deliverable software in the development of deliverable software as long as the operation and support

- Software installation planning: The developer shall develop and record plans for performing software installation and training at the user sites specified in the contract. This planning shall include all applicable items in the Software Installation Plan (SIP).
- Software transition planning: The developer shall identify all software development resources that will be needed by the support agency to fulfill the support concept specified in the contract. The developer shall develop and record plans identifying these resources and describing the approach to be followed for transitioning deliverable items to the support agency. This planning shall include all applicable items in the Software Transition Plan (STrP).
- Following and updating plans: Following acquirer approval of any of the plans in this section, the developer shall conduct the relevant activities in accordance with the plan. The developer's management shall review the software development process at intervals specified in the software development plan to assure that the process complies with the contract and adheres to the plans. With the exception of developer-internal scheduling and related staffing information, updates to plans shall be subject to acquirer approval.

Establishing a software development environment

The developer shall establish a software development environment in accordance with the following requirements.

* **Note**: If a system or CSCI is developed in multiple builds, establishing the software development environment in each build should be

c) Permit representations other than traditional documents for recording the information (e.g., computer-aided software engineering (CASE) tools).

* **Note 2**: If the CDRL specifies delivery of the information generated by this or any other paragraph, the developer is required to format, assemble, mark, copy, and distribute the deliverable in accordance with the CDRL. This task is recognized to be separate from the task of generating and recording the required information and to require additional time and effort on the part of the developer.
* **Note 3**: The software development plan covers all activities required by this standard. Portions of the plan may be bound or maintained separately if this approach enhances the usability of the information. Examples include separate plans for software quality assurance and software configuration management.

- CSCI test planning: The developer shall develop and record plans for conducting CSCI qualification testing. This planning shall include all applicable items in the Software Test Plan (STP).
- System test planning: The developer shall participate in developing and recording plans for conducting system qualification testing. For software systems, this planning shall include all applicable items in the Software Test Plan (STP). The intent for software systems is a single software test plan covering both CSCI and system qualification testing.

- Joint technical and management reviews
- Other activities

Project planning and oversight

The standard states that the developer shall perform project planning and oversight in accordance with the following requirements.

Note: If a system or CSCI is developed in multiple builds, planning for each build should be interpreted to include: (a) overall planning for the contract, (b) detailed planning for the current build, and (c) planning for future builds covered under the contract to a level of detail compatible with the information available.

- Software development planning: The developer shall develop and record plans for conducting the activities required by this standard and by other software-related requirements in the contract. This planning shall be consistent with system-level planning and shall include all applicable items in the Software Development Plan (SDP).

  * **Note 1**: The wording here and throughout MIL-STD-498 standard referenced document [STD94] is designed to:

    a) Emphasize that the development and recording of planning and engineering information is an intrinsic part of the software development process, to be performed regardless of whether a deliverable is required;

    b) Use the DID as a checklist of items to be covered in the planning or engineering activity; and

The following table (Table 2.2)summarizes the major differences between the above models.

Life Cycle Model	Define All Requirements First?	Multiple Development Cycles?	Field Interim Products?
Sequential	Yes	No	No
Incremental	Yes	Yes	May be
Evolutionary	No	Yes	Yes

Table 2.2: Key features of selected life cycle model

2.3.2.3 Management activities

In the figures describing the life cycle models in the previous section several management activities were shown. These activities span the whole development effort as shown at the top and the bottom parts of the Figure 2.9 and Figure 2.10. A brief description of the management activities is given in the following paragraphs of this section. These activities are listed as follows:

- Project planning and oversight
- Establishing a software development environment
- Software configuration management
- Software product evaluation
- Software quality assurance
- Corrective “action”

)

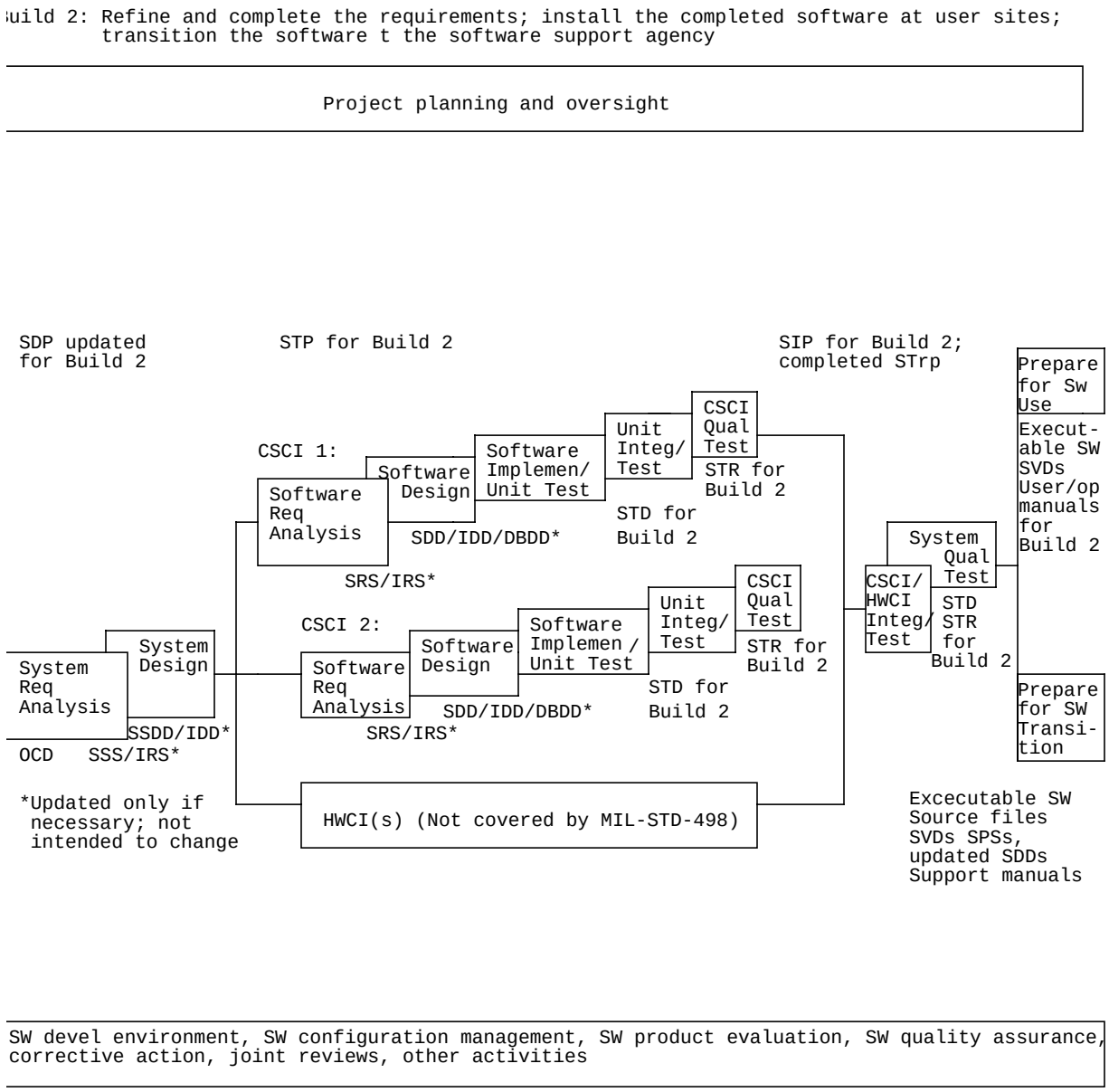


FIGURE 2.12 Evolutionary Model *The phases of Build 2* (Adapted from MIL-STD 498)

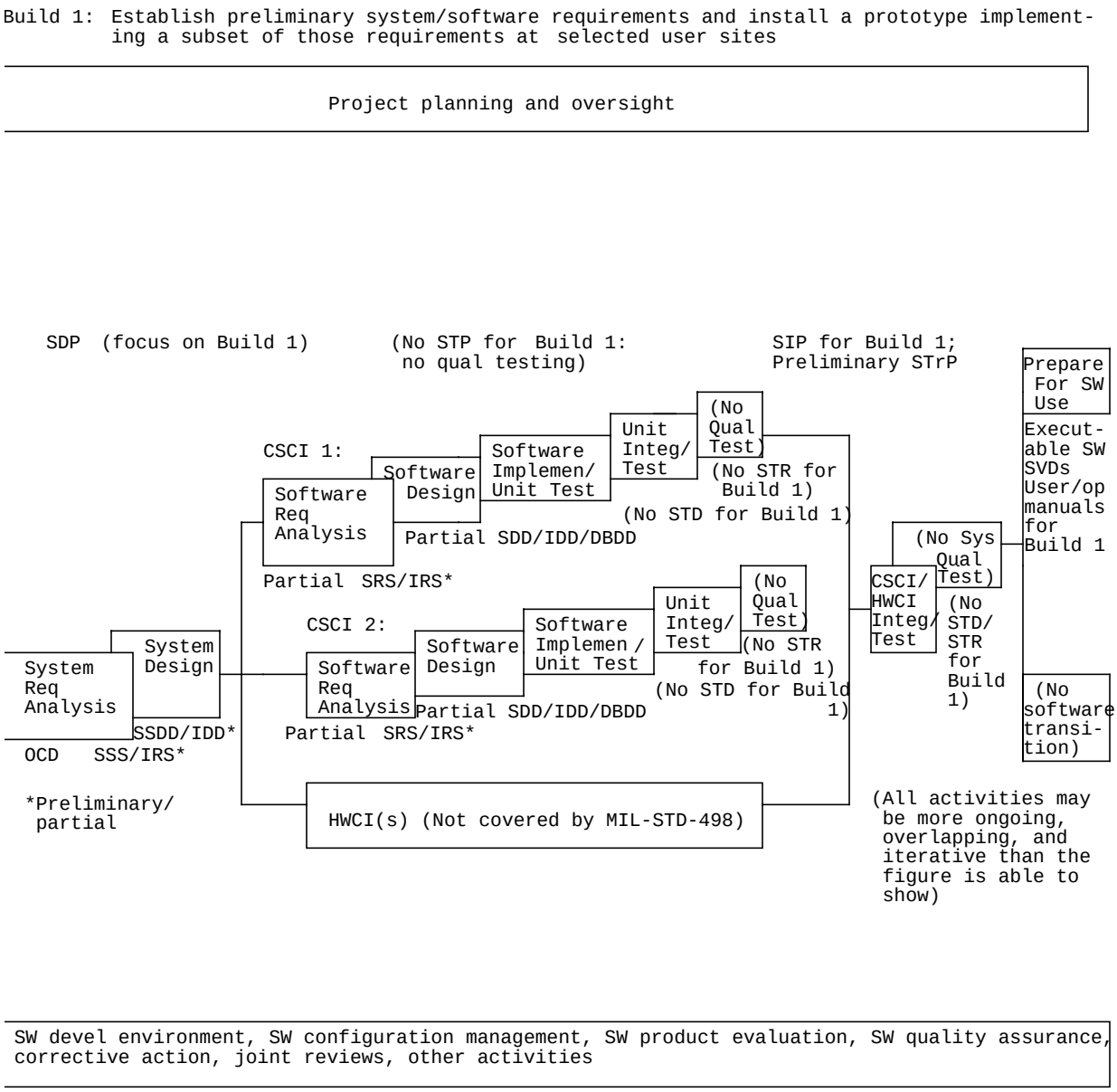


FIGURE 2.11 Evolutionary Model *The phases of Build 1*(Adapted from MIL-STD 498

- Evolutionary development: The evolutionary model also develops a system in builds, but differs from the incremental model in acknowledging that the user need is not fully understood and all requirements cannot be defined up front. In this model, user needs and system requirements are partially defined up front. With each succeeding build, these requirements get more and more refined. Figure 2.11 and Figure 2.12 show the development phases in this model for Build 1 and Build 2 respectively. As shown in the figures, even the first three phases are part of the development iterative cycle. Therefore, the documents produced in these phases are labeled as preliminary or partial since they are refined in each succeeding build. Moreover, the build produced for a given CSCI in each iterative cycle is released and is integrated and tested with the other components in the last four phases of development.

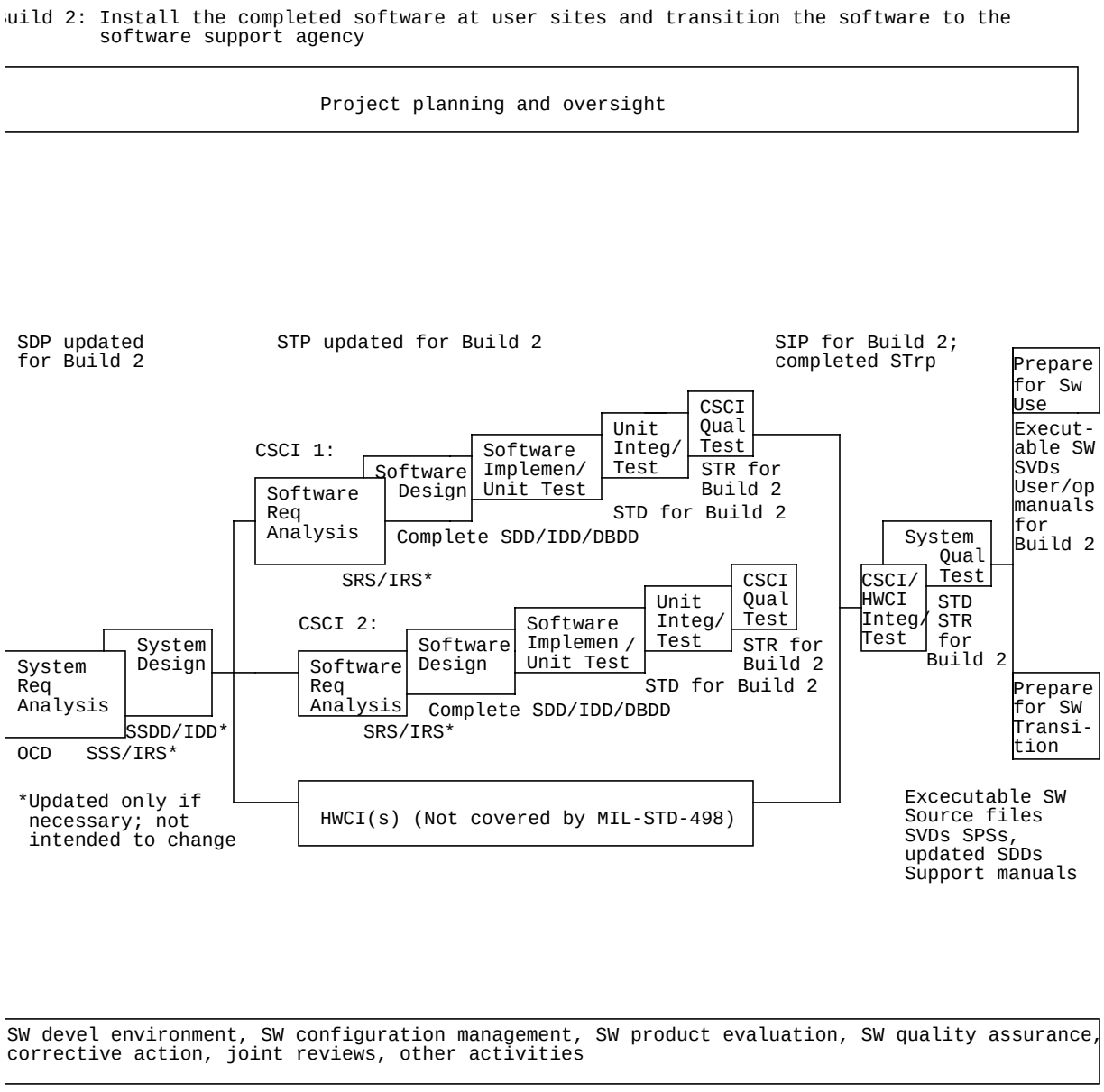


FIGURE 2.10 Incremental Model *The phases of Build 2* (Adapted from MIL-STD 498)

phases of the model will be purely sequential similar to those in sequential model.

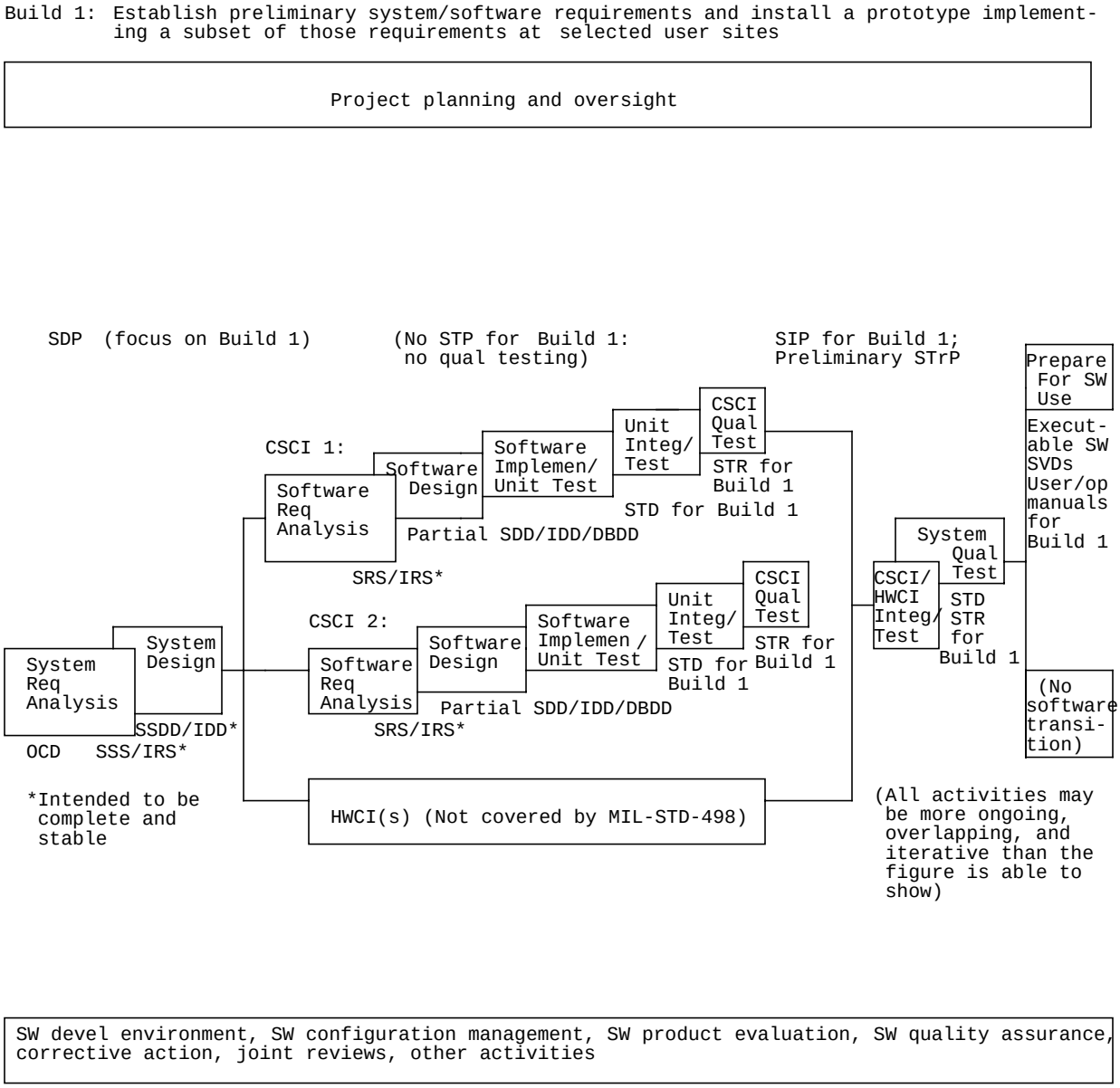


FIGURE 2.9 Incremental Model *The phases of Build 1* (Adapted from MIL-STD 498)

These models are also used here to discuss possible candidate life cycle models for this standard. The sequential model has already been described earlier as shown in Figure 2.8.

The other two models namely Incremental Development Model and the Evolutionary Model are discussed in the following paragraphs.

- Incremental development: The “incremental” model determines user needs and defines the system requirements, then performs the rest of the development in a sequence of builds. The first build incorporates part of the planned capabilities. The next build adds more capabilities, and so on, until the system is complete. Figure 2.9 and Figure 2.10 show the development phases in this model for Build 1 and Build 2 respectively. The first three phases, namely the SYSRA phase, SYSD phase, and the SWRA phase for a given CSCI are sequential and are exactly similar to those in the sequential model. The difference lies in the next four phases (starting from SWD until the CSCIQT). As opposed to the sequential model, these four phases are iterative. In each iteration a particular build for the CSCI is produced. This build depending on its capabilities may or may not be integrated with other CSCI builds and HWCI’s. In this case the last four phases (starting from CSCI/HWCI Integration testing to the PSWT phase) will also be part of the iterative development process. This will result in multiple releases corresponding to the multiple builds as shown in Figure 2.9 and Figure 2.10. The model described in these figures is similar to the incremental delivery model described with the ESA standard. Incremental development, however, is more general since the CSCI builds can be integrated with the HWCI’s only when they are fully complete. In which case the last four

- Demonstrate to the acquirer that the deliverable software can be regenerated (compiled/linked/loaded into an executable product). Deliverables should be maintained using commercially available, acquirer- owned, or contractually deliverable software and hardware designated in the contract or approved by the acquirer.
- Provide training to the support agency as specified in the contract.
- Provide other assistance to the support agency as specified in the contract.

2.3.2.2 The Life-Cycle Models

The waterfall model used in the previous section to illustrate the activities required by the 498 standard is not a representative of what the standard is advocating (as its predecessor, the 2167A standard, seemingly did). In fact the 498 standard does not recommend any particular life cycle model. It emphasizes that the above activities may overlap, may begin and end in any order, may be applied iteratively, and may be applied differently to different elements of the software. This statement concerning the order of these activities can only be explained by discussing candidate life cycle models encapsulating these activities.

Three types of life cycle models are discussed with the ESA standard in the previous section. These are:

- the sequential development approach,
- the incremental development approach, and
- the evolutionary development approach.

- Update the design description of each CSCI to match the “as built” software and shall define and record:

  a) the methods to be used to verify copies of the software,

  b) the measured computer hardware resource utilization for the CSCI,

  c) other information needed to support the software,

  d) traceability between the CSCI’s source files and software units and

  e) traceability between the computer hardware resource utilization measurements and the CSCI requirements concerning them.

  The result shall be recorded in the qualification, software support, and traceability sections of the Software Product Specification (SPS). The “as built” design is included in the software product specification not as the product but as information that may help the support agency understand the software in order to modify, enhance, and otherwise support it.

- Participate in updating the system design description to match the “as built” system. The result shall be recorded in the SSDD document.

- Identify and record information needed to program the computers on which the software was developed or on which it will run. The information shall be recorded in the Computer Programming Manual (CPM).

- Identify and record information needed to program and reprogram any firmware devices in which the software will be installed. The information shall be recorded in the Firmware Support Manual (FSM).

- Install and check out the executable software at the user sites specified in the contract.
- Provide training to users as specified in the contract.
- Provide other assistance to user sites as specified.

The above manuals are not required for every project. As mentioned in the standard that few, if any, systems will need all of the manuals listed above. The intent is for the acquirer, with input from the developer, to determine which manuals are appropriate for a given system and to require the development of only those manuals. The manuals are normally developed in parallel with software development, ready for use in CSCI testing.

K) Preparing for Software Transition (PSWT)

The developer shall prepare the executable software to be moved to the support site, including any batch files, command files, data files, or other software files needed to install and operate the software on its target computer(s). The result shall be recorded in the executable software section of the Software Product Specification (SPS). In addition, the specific requirements are as follows:

- Prepare the source files to be transitioned to the support site, including any batch files, command files, data files, or other files needed to regenerate the executable software. The result shall be recorded in the source file section of the Software Product Specification (SPS).
- Identify and record the exact version of software prepared for the support site. The information shall be recorded in the Software Version Description (SVD) document.

J) Preparing for Software Use (PSWU)

The developer shall prepare the executable software for each user site, including any batch files, command files, data files, or other software files needed to install and operate the software on its target computer(s). In addition, the specific requirements in this phase include the following:

- Identify and record the exact version of software prepared for each user site. The information shall be recorded in the Software Version Description (SVD).
- Identify and record information needed by hands-on users of the software (persons who will both operate the software and make use of its results). The information shall be recorded in the Software User Manual (SUM).
- Identify and record information needed by persons who will submit inputs to, and receive outputs from, the software, relying on others to operate the software in a computer center or other centralized or networked software installation. The information shall be recorded in the Software Input/Output Manual (SIOM).
- Identify and record information needed by persons who will operate the software in a computer center or other centralized or networked software installation, so that it can be used by others. The information shall be recorded in the Software Center Operator Manual (SCOM).
- Identify and record information needed to operate the computers on which the software will run. The information shall include all applicable items in the Computer Operation Manual.

- Participate in developing and recording the test preparations, test cases, and test procedures to be used for system qualification testing and the traceability between the test cases and the system requirements. For software systems, the results shall include all applicable items in the Software Test Description (STD) document. The developer shall participate in preparing the test data needed to carry out the test cases and in providing the acquirer advance notice of the time and location of system qualification testing.
- Participate in dry running the system test cases and procedures to ensure that they are complete and accurate and that the system is ready for witnessed testing. The developer shall record the software-related results of this activity in appropriate software development files (SDFs) and shall participate in updating the system test cases and procedures as appropriate.
- Participate in system qualification testing. This participation shall be in accordance with the system test cases and procedures.
- Make necessary revisions to the software, provide the acquirer advance notice of retesting, participate in all necessary retesting, and update the software development files (SDFs) and other software products as needed, based on the results of system qualification testing.
- Participate in analyzing and recording the results of system qualification testing. For software systems, the result shall be documented in the Software Test Report (STR).

- Make necessary revisions to the software, participate in all necessary retesting, and update the appropriate software development files (SDFs) and other software products as needed, based on the results of CSCI/HWCI integration and testing.
- Participate in analyzing the results of CSCI/HWCI integration and testing. Software-related analysis and test results shall be recorded in appropriate SDFs.

I) System Qualification Testing (SYSQT)

System qualification testing is performed to demonstrate to the acquirer that system requirements have been met. It covers the system requirements in the system/subsystem specifications (SSSs) and in associated interface requirements specifications (IRSs). This testing contrasts with developer-internal system testing, performed as the final stage of CSCI/HWCI integration and testing.

The developer's system qualification testing must include testing on the target computer system or an alternative system approved by the acquirer.

The person(s) responsible for fulfilling the requirements in this section shall not be the persons who performed detailed design or implementation of software in the system. This does not preclude persons who performed detailed design or implementation of software in the system from contributing to the process, for example, by contributing test cases that rely on knowledge of the system's internal implementation.

The following specific activities are required:

- Perform CSCI qualification testing of each CSCI. The testing shall be in accordance with the CSCI test cases and procedures.
- Make necessary revisions to the software, provide the acquirer advance notice of retesting, conduct all necessary retesting, and update the SDFs and other software products as needed, based on the results of CSCI qualification testing.
- Analyze and record the results of CSCI qualification testing in the Software Test Report (STR).

H) CSCI/HWCI Integration and Testing (IT)

CSCI/HWCI integration and testing means integrating CSCIs with interfacing HWCIs and other CSCIs, testing the resulting groupings to determine whether they work together as intended. This process continues until all CSCIs and HWCIs in the system are integrated and tested. The last stage of this testing is developer-internal system testing. The software developer is only required to participate in the activities in phase with other players. The following specific activities are required:

- Participate in developing and recording test cases (in terms of inputs, expected results, and evaluation criteria), test procedures, and test data for conducting CSCI/HWCI integration and testing. The test cases shall cover all aspects of the system-wide and system architectural design. The developer shall record software-related information in SDFs.
- Participate in CSCI/HWCI integration and testing. The testing shall be in accordance with the CSCI/HWCI integration test cases and procedures.

covers the CSCI requirements in software requirements specifications (SRSs) and in associated interface requirements specifications (IRSs). This testing contrasts with developer-internal CSCI testing, performed as the final stage of the previous phase.

The standard requires the CSCI qualification testing to be independent by having different persons (i.e., not from the developing team) perform the activities of this test. This means that person(s) responsible for qualification testing of a given CSCI shall not be the persons who performed detailed design or implementation of that CSCI. This does not preclude persons who performed detailed design or implementation of the CSCI from contributing to the process, for example, by contributing test cases that rely on knowledge of the CSCI's internal implementation. CSCI qualification testing shall include:

- Testing on the target computer system or an alternative system approved by the acquirer.
- Define and record the test preparations, test cases, and test procedures to be used for CSCI qualification testing and the traceability between the test cases and the CSCI requirements. The result shall include all applicable items in the Software Test Description (STD). The developer shall prepare the test data needed to carry out the test cases and provide the acquirer advance notice of the time and location of CSCI qualification testing.
- Inspect and perform as a "dry run" (i.e., before hand) the CSCI test cases and procedures to ensure that they are complete and accurate and that the software is ready for witnessed testing. The developer shall record the results of this activity in appropriate SDFs and shall update the CSCI test cases and procedures as appropriate.

- Test the software corresponding to each software unit. The testing shall be in accordance with the unit test cases and procedures.
- Make all necessary revisions to the software, perform all necessary retesting, and update the SDFs. Based on the results of the unit testing, other software products should be revised.
- Analyze the results of unit testing and record the test and analysis results in appropriate SDFs.

F) Unit Integration and Testing (UIT)

Unit integration and testing involves integrating the software corresponding to two or more software units, testing the resulting software to ensure that it works together as intended. This process continues until all the software in each CSCI is integrated and tested. The last stage of this testing is developer-internal CSCI testing. Since units may consist of other units, some unit integration and testing may take place during unit testing. The activities required in this phase are not meant to duplicate those activities. The order of activities performed are similar to the order of activities in the unit testing as follows:

- establish test cases and test procedures for unit integration and testing,
- perform the testing,
- make necessary revisions and perform retesting, and
- analyze as well as record unit integration and test results in appropriate SDFs.

G) CSCI Qualification Testing (CSCIQT)

CSCI qualification testing is performed to demonstrate to the acquirer (i.e., the contracting agency and their users and consultants) that the CSCI requirements have been met. It

- Define and record the architectural design of each CSCI, identifying the software units comprising the CSCI, their interfaces, and a concept of execution among them,
- Show the traceability between the software units and the CSCI requirements.

The result of the effort in this phase is documented in the Software Design Description (SDD) document. Design pertaining to interfaces may be included in SDDs or in Interface Design Descriptions (IDD) document.

E) Software Implementation and Unit Testing (SWIUT)

The software implementation and unit testing (SWIUT) phase is similar to the coding and unit testing activities in the DD phase of the ESA standard. The 498 standard requires the following activities:

- Develop and document software corresponding to each software unit in the CSCI design. This activity includes building databases, populating databases and other files with data values, and other activities needed to implement the design. For deliverable software, the developer shall obtain acquirer approval to use any programming language not specified in the contract.
- Establish test cases (in terms of inputs, expected results, and evaluation criteria), test procedures, and test data for testing the software corresponding to each software unit. The test cases shall cover all aspects of the unit's detailed design. The developer shall record this information in the appropriate Software Development Files (SDFs).

C) Software Requirements Analysis (SWRA)

In the SWRA phase the developer is required to specify and record the software requirements to be met by each CSCI. The activities in this phase are similar to those specified in the SR phase of the ESA standard. Besides specifying the software requirements, the following activities are also required by the 498 standard:

- Define the methods to be used to ensure that each requirement has been met,
- Show the traceability between the CSCI requirements and the system requirements,
- Define and record a complete set of interface requirements for each interface external to every CSCI.

The result of the SWRA phase is documented in the Software Requirements Specification (SRS) document. Requirements concerning CSCI interfaces may be included in SRSs or in the Interface Requirements Specifications (IRSs) document.

D) Software Design (SWD)

The SWD phase consists of the activities similar to those specified in the AD phase of the ESA standard. The following specific activities are required by the 498 standard:

- Define and record CSCI-wide design decisions (that is, decisions about the CSCI's behavioral design and other decisions affecting the selection and design of the software units comprising the CSCI). For example, a decision to use a set of reusable software components will be a decision that affects the selection and design of software units comprising the CSCI,

design of system components. Participation of the developer is also needed to define and record the architectural design of the system. Traceability between the system components and system requirements should be established. The following are the two major activities in this phase:

- Participate in defining and recording system-wide design decisions (i.e., decisions about the system's behavioral design and other decisions affecting the selection and design of system components).
- Participate in defining and recording the architectural design of the system (identifying the components of the system, their interfaces, and a concept of execution among them) as well as the traceability between the system components and system requirements.
- The result of the SYSD phase is the System/Subsystem Design Description (SSDD). The design pertaining to interfaces may be included in the SSDD or in an Interface Design Description (IDD) document. Design pertaining to databases may be included in the SSDD or in a Database Design Descriptions (DBDDs) document.
- The activities outlined above in SYSD phase should define and record the overall system appearance and behavior in response to system requirements as well as the allocation of system requirements to Hardware Configuration Items (HWCIs), Computer Software Configuration Items (CSCIs), and manual operations.

The SYSRA phase is different from the UR phase of the ESA standard described in the previous section. The scope of the SYSRA phase in the 498 standard is the system as whole while the ESA standard only addresses the software, during the UR phase. Therefore, the software developer is only required to participate in the activities of this phase with other players such as the contracting organization, and perhaps other contractors.

The following are the specific requirements on the activities in this phase:

- Participate in analyzing user input provided by the acquirer to gain an understanding of user needs. This input may take the form of need statements, surveys, problem/change reports, feedback on prototypes, interviews, or other user input or feedback.
- Participate in defining and recording the operational concept for the system. The result shall include all applicable items in the Operational Concept Description (OCD).
- Participate in defining and recording the requirements to be met by the system and the methods to be used to ensure that each requirement has been met. The result shall include all applicable items in the (SSS). Depending on CDRL provisions, requirements concerning system interfaces may be included in the SSS or in interface requirements specifications (IRS).

B) System Design (SYSD)

In the SYSD phase, the developer is required to participate (again with the other players) in defining and recording the system-wide design decisions affecting the selection and

- Software Test Report (STR).

The parallel threads of development merge into a sequential development thread containing the two phases of IT and SYSQT where the system STD and STR documents are produced. The last two phases of PSWU and PSWT are accomplished in parallel and the following final documents are produced:

- executable code,
- Software Product Specification (SPS) document;
- Software Version Description (SVD) document;
- Software User Manual (SUM);
- Software Input/Output Manual (SIOM);
- Software Center Operator Manual (SCOM), and
- other support manuals.

2.3.2.1 MIL-STD-498 Software Development Phases

The following paragraphs briefly describe the activities in the each of the above phases.

A) System Requirements Analysis (SYSRA)

In the SYSRA phase, the users input is surveyed and analyzed, the operational concept of the system is defined and documented in the Operational Concept Description Document (OCD). The system requirements are documented in a System/Subsystem Specification (SSS) document, and the interfaces' requirements are specified and recorded in an Interface Requirement Specifications (IRS) document.

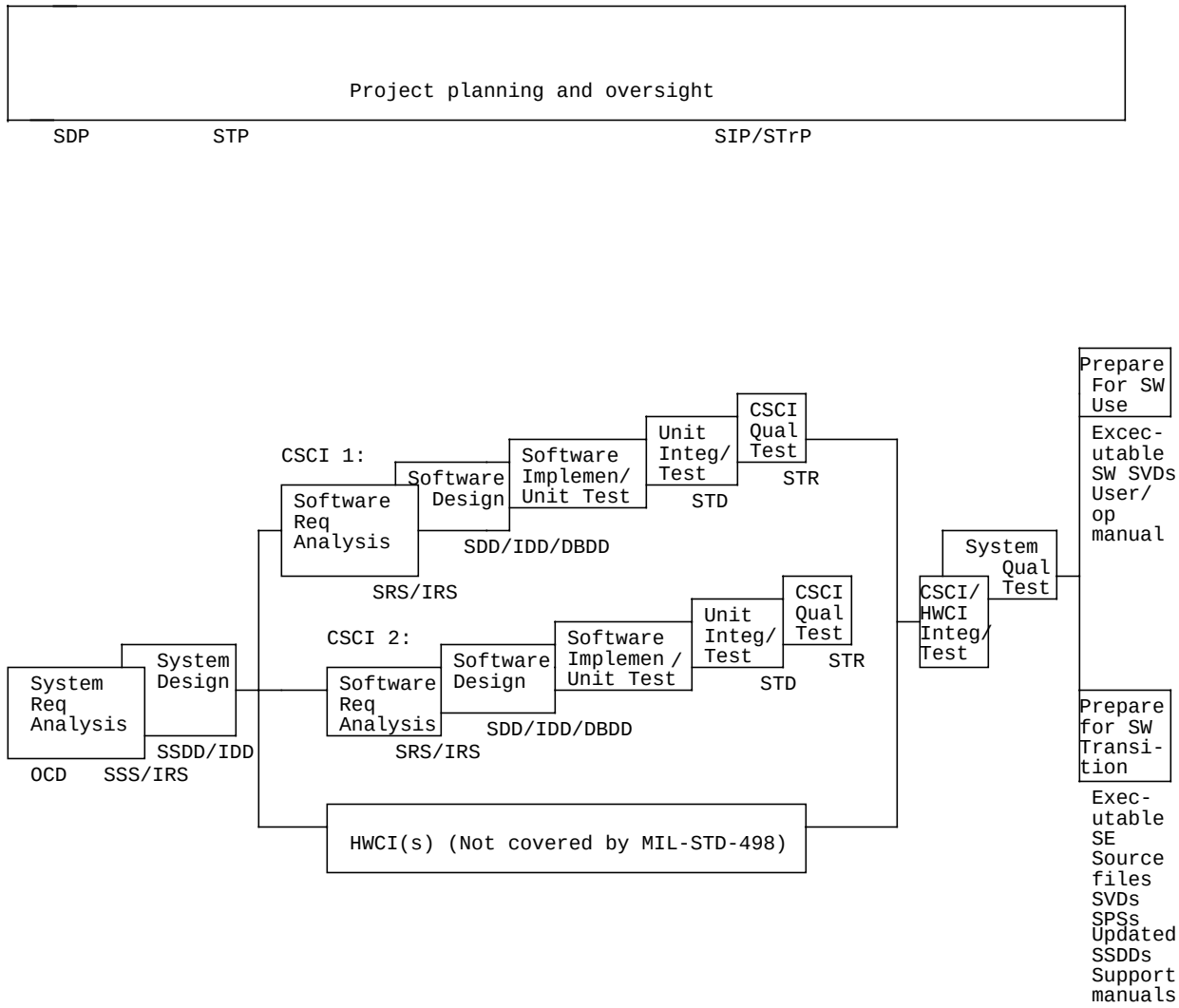
Models based on the incremental development approach and the evolutionary development approach for software development will be discussed later in this section.

Figure 2.8 also shows a sequential process in the beginning where the SYSRA and SYSD phases are accomplished and the following documents are produced:

- Operational Concept Description Document (OCD),
- the System/Subsystem Specification (SSS) document,
- the Interface Requirement Specification (IRS) document,
- the System/Subsystem Design Description (SSDD) document, and
- the Interface Design Description (IDD) document.

Parallel development threads are then shown for the CSCIs and HWCI's specified in the SSDD document. For each CSCI, the sequence of the SWRA phase -to- a SWD phase -to- a SWIUT phase -to- a CSCIQT phase is accomplished. These phases produce the following sequence of documents:

- Requirements Specification (SRS) document. Requirements concerning CSCI interfaces may be included in SRSs or in the Interface Requirements Specifications (IRSs) document;
- Software Design Description (SDD), design pertaining to interfaces may be included in SDDs or in Interface Design Descriptions (IDDs) and design pertaining to databases may be included in SDDs or in Database Design Descriptions (DBDDs);
- Software Test Description (STD); and



SW devel environment, SW configuration management, SW product evaluation, SW quality assurance, privacy, interface with IV&V, coordination with associate developers, improvement of project processes.

Note: All activities may be more ongoing, overlapping, and iterative, than the figure is able to show.

FIGURE 2.8 Sequential Development Model (Adapted from MIL-STD 498)

together with a discussion on software life-cycle models similar to those described for ESA standard. The management activities are described later in the section.

Software engineering activities

The general requirements specified by the standard for the software engineering phases are summarized below. These phases result in documents called Data Item Descriptions (DIDs). The requirements for the preparation of the various sections in the analysis and design documents are given in Chapters 3 and 4 respectively. The software engineering process shall include the following major activities.

- System requirements analysis (SYSRA)
- System Design (SYSD)
- Software Requirements Analysis (SWRA)
- Software Design (SWD)
- Software Implementation and Unit Testing (SWIUT)
- Unit Integration and Testing (UIT)
- CSCI Qualification Testing (CSCIQT)
- CSCI/HWCI Integration and Testing (IT)
- System Qualification Testing (SYSQT)
- Preparing for Software Use (PSWU)
- Preparing for Software Transition (PSWT)

Figure 2.8 shows a possible development model encapsulating the above phases. This model is a sequential development model similar to the waterfall model described before.

- provides more emphasis on software supportability, and
- improves links to systems engineering.

This section contains excerpts of this standard, edited and explained to conform with the format of this text.

The software to which the standard is applied may or may not be designated as Computer Software Configuration Items (CSCIs). While this standard is written in terms of CSCIs, it may be applied to any software, with the term “CSCI” interpreted appropriately. A Computer Software Configuration Item (CSCI) is defined here as an aggregation of software that satisfies an end use function and is designated by the government for separate configuration management. CSCIs are selected based on many factors, including:

- trade-offs among software function,
- size,
- host or target computers,
- developer,
- support concept,
- plans for reuse,
- criticality,
- interface considerations,
- need to be separately documented and controlled.

In the following paragraphs, an overview of the standard will be presented. The term contractor refers to the software developer.

The 498 standard activities will be described here in a similar manner as the ESA standard is described in the previous section. The software development activities will be described first

- Producing a Software Configuration Management Plan (SCMP): This plan documents all the activities mentioned above. Plans for each phase of the life cycle must be outlined.

The above activities are applied in every phase of life-cycle on deliverables such as documentation, CASE tools outputs, prototype code, or deliverable code.

2.3.2 The MIL-STD-498 Software Development Standard

The Department of Defence (DOD) in the US government has a history of sponsoring the development of standards to be used by all its departments and agencies for both the external as well as in-house software development projects. Efforts in the early 80's resulted in the development of the DOD-STD-2167A standard. This standard (DOD 2167A) is based on the waterfall model for software development and has been used for several years in various government sponsored projects. The documentation structure of the this standard has also been supported by ICASE tools such as Teamwork. Recently a new standard has been approved to supersede the above standard and merge it with the DOD-STD-7935A (standard for Automated Information systems documentation). The new standard, namely the MIL-STD-498, will be referred to here as the 498 standard. For the complete specifications of the 498 standard the reader is urged to consult the reference document [STD94].

This standard improves compatibility with incremental and evolutionary software development process models, with non- hierarchical analysis and design methods such as the object- oriented approach, and with ICASE tools. In addition, the standard has the following features:

- it adds clear requirements for incorporating reusable software,
- introduces the use of software management indicators,

The standard mandates that all software items, including documentation, tools, test software, and test data, must be subject to Configuration Management (CM) procedures. The CM should be clearly specified using a set of common procedures. The activities specified for configuration management are briefly specified as follows:

- Configuration identification: Each component specified during the design phase must be identified as a configuration item (CI). The identification of a CI must include a name related to the function of the CI, a processing type indicator and a version number.
- Establishing software libraries: The standard mandates that three types of software libraries must be implemented to support the storage of the deliverable items. These libraries include a development library for CIs under development or under change; a Master Library for tested CIs under integration and system testing; and a static or archive library consisting of released or baselined Master libraries of CIs which must not be changed.
- Configuration control: The process of deciding on proposed changes to CIs and controlling the implementation of approved changes is termed as software configuration change control. Changing delivered documents must undergo a rigorous change procedure of drafts and reviews which is documented in a Document Change Record (DCR). During these reviews, Review Item Discrepancy (RID) forms are filled and responded to.
- Configuration status accounting: Each CI status must be recorded with information including description of each item, date of release, and date or status of RIDs.

- Planning and organizing the development activities in each phase discussed above of the product life cycle. This include defining team structures and responsibilities, estimating resources and time schedules and budgeting.
- Risk analysis, where project risks should be identified and assessed. Risk factors would result in the software being delivered late and/or over budget. These factors are either user-specific (such as unstable user requirements or systems interfaces), or developer-specific (such as availability of resources, tools, and experienced staff).
- Managing the technical aspects of the project such as deciding on methods and tools to be used for product development and process management throughout the life cycle.
- Preparation of progress reports and conducting progress reviews. These progress reports should be detailed enough to be included in the Project History Document (PHD).
- Producing the Software Project Management Plan (SPMP). This plan must document all the management activities throughout the life cycle. The SPMP documents at the end of the UR phase the outline of the plan for the whole project and a detailed plan for the activities in the SR phase. Similarly at each phase, detailed plans for the following phase must be documented. The project cost must be estimated in the UR phase and refined in each successive phase. During the AD phase, the SPMP must include a Work Breakdown Structure (WBS) which corresponds to the software decomposition into components.

Software Configuration Management

The Software Transfer Document (STD) specifies the transferred software, how to build it, and how to install it. The STD summarizes the acceptance test reports, and all the software change documents, produced during the TR phase.

The Operations and Maintenance Phase

At the end of the TR phase, a statement of provisional acceptance must be made on behalf of the users and delivered to the developer. It is during the OM phase that the software enters operational use. Final acceptance tests must be conducted, and the developer should still be responsible for correcting operational errors. A maintenance team or organization must be selected. The maintenance of the software is conducted when problems occur. It is strongly recommended that users be informed of the problem. If possible, software should be taken out of operation until the problem is corrected.

The outputs of the TR phase consist of the final acceptance statement delivered to the developer. The developer should produce a Project History Document (PHD) containing resources required and both the estimated cost in the beginning of the project and the measured cost at the end of the project.

2.3.1.2 ESA Procedure Standards

The procedure standards part describes the procedures used to manage the software development project. These include the standards for software project management, software configuration management, software verification and validation, and software quality assurance. The first two will be described briefly in the following paragraphs.

Software Project Management

The main activities of project management cited are summarized as follows:

activities must be documented in the software verification and validation plan to be described in Chapter 6 of this text.

The outputs of the DD phase consist of the Detailed Design Document (DDD), the code, the software user manual (SUM), and the plans for the transfer (TR) phase. Other documents such as progress, verification, and audit reports and configuration status accounts for the project should also be archived. The first part of the DDD documents the detailed design standards, and coding conventions and the tools to be used. This part of the document is to be prepared as the first activity in the DD phase before the detailed design and coding starts. The second part of the DDD must have the same section structure documenting the code as the component structure of the code itself. Traceability information between software requirements in the SRD and detailed design components must be specified in the DDD.

The Transfer Phase

The activities performed in the TR phase consists of installation of the acceptance testing, provisional acceptance, and the development of the software transfer document. Software installation is accomplished by building an executable system in the target environment. It is required that the maintenance team be able to build an executable system from modifiable system components. In fact it is strongly recommended that the maintenance team exercise the modification of components and the installation of the software. Acceptance testing is based on validating the capabilities of the software, based on the user requirements documented in the URD. The test plans and test designs and test cases and test procedures are specified in the software verification and validation plan (SVVP).

The Detailed Design and Production Phase

The major activities in this phase include top-down decomposition and the specification of the low-level software modules, followed by software production and documentation using structured programming. These activities are:

a) Decomposition of low-level components

b) Software production

These are briefly summarized as follows.

A) Decomposition of low-level components

Using step-wise refinement, the low-level components specified at the architectural design are decomposed into software modules. A software module is defined as an independently testable program unit with well defined inputs and outputs which can be separately compiled and linked with other units. The standard recommends that each module specification be reviewed and approved for coding.

B) Software production

Software production consists of implementing the modules specifications into code, integrating the software components, and the testing activities. Coding conventions (e.g., comments structure, naming of subprograms and variables, and error handling) are strongly recommended. Structured programming rules to be followed during coding are also recommended. Integrating the coded components was mandated to be controlled by the software configuration management procedure which is described in the procedure standards. Finally the testing activities in terms of unit testing, integration testing, and system testing are also part of the detailed design phase. All testing procedures and

B) Specifying the architectural design

The architectural design should be specified using diagrams showing the components at each level of the hierarchy and the data and control coupling between them. The data structures of each data coupling must be specified in the Architectural Design Document (ADD). The control flows between components must also be defined in the ADD. It is also required that for each component, the data input, data output, and the operations performed by the component must be clearly specified. Finally, the computing resources needed in terms of CPU time, memory space must be estimated and documented in the ADD.

The outputs of the AD phase consist of the ADD, the integration test plan, and the management plans for the Detailed Design (DD) phase. The management plans will be briefly described later this section.

The reviews in the AD phase consist of:

- strongly recommended reviews of the architectural design as it develops level by level and
- the mandatory formal review of the outputs of the AD phase, i.e., the Architectural Design Review (AD/R).

The AD/R is very important since it is very costly to change the architectural design after the DD phase has been started. In fact the standard strongly recommends that the DD phase should not start if there are any doubts in the architectural design. Walkthroughs of the architecture design is strongly recommended during the AD phase to eliminate doubts and problems in the design as they appear.

The most important output documents of the SR phase are the SRD and system test plan. The SRD should document all the activities described above. The system test plans specify the techniques and approaches to be used in validating the software implementation. Several types of management plans for the AD phase are also produced in this phase. The software requirements review (SR/R) must be formally conducted to review the output documents of the SR phase.

The Architectural Design (AD) Phase

The major activities specified in the AD phase are:

a) Developing the software architecture

b) Specifying the architectural design

These are summarized as follows.

A) Developing the software architecture

In this activity, the physical software architecture is developed from the logical model specified in the SR phase. The architecture is developed as a hierarchy of software components or software modules. The development method used, must permit a top-down design approach in order to control complexity. The standard strongly recommends the use of design quality criteria such as design efficiency understandability and adaptability. Modular design is also strongly recommended. As mentioned before, modular design stands for having a design with minimum coupling between modules and maximum cohesion within each module. Although ADD specifies the selected design only, trade-off criteria and reasons for choosing this design should be documented in a Project History Document.

- Safety requirements specify the system-specific constraints needed to reduce the damage resulting from a software failure.
- Attributes are also attached to each requirement. These attributes are exactly similar to the ones discussed above for the user requirements during the UR phase.

C) Analyzing the software requirements

The objective of the requirements analysis activity is to insure the completeness and consistency of the software requirements. Completeness is achieved when each user requirement specified in the URD is accounted for, and there are no “to be defined” (or TBD) terms in the software requirement document. A traceability matrix between requirements in the URD and requirements in the SRD is specified to prove completeness. Consistency, on the other hand, is achieved when conflicting sets of requirements are not found. These conflicts can be in many hidden forms such as:

- using different terms in different requirements, which basically refer to the same item, or
- using the same general term to refer to different items;
- requirements specifying incompatible activity happening in the same time or
- activities happening in the wrong order.

Finally, duplicating requirements should also be avoided. However, if it is necessary to duplicate requirements for the sake of understandability, cross references must be inserted to enhance modifiability and eliminate the possibility of modifying a requirement without modifying its duplicates.

- Operation requirements include those requirements that specify the usability, or the user interface of the software system.
- Resource requirements specify constraints on the available resources such as memory size or disk space.
- Documentation requirements specify the project-specific documentation requirements such as a specific format for developing the user interface and the user documentation in general.
- Security requirements relate to the system-specific protection components, such as a sophisticated password component, a virus detection component, etc.
- Quality requirements should specify in measurable terms, using metrics, as much as possible, the quality of analysis, design, and implementation of the system.
- Verification requirements specify the methodologies in which the software must be verified such as requirements for using simulation models, live test with simulated input, or with real inputs.
- Reliability requirements, specify the system reliability measure in terms of Mean Time Between Failures (MTBF) as well as the minimum time between failures.
- Maintainability requirements specify the system maintainability measure in terms of Mean Time To Repair (MTTR) and may include methodology constraints specified by maintenance groups.

The above rules will be demonstrated by examples in Chapter 3. The second activity in the SR phase is described as follows:

B) Specifying the software requirements

The standard strongly recommends that requirements be rigorously specified, and if possible should be stated in quantitative terms. The logical model developed in the previous activity is used to specify and classify requirements. Requirements are classified as functional, performance, interface related, operational, resource related, verification, acceptance testing, documentation related, security related, portability related, quality, reliability, maintainability, and safety related requirements. These types are of particular importance for hard, real-time systems. The above requirements classes are explained briefly as follows:

- Functional requirements define the operations the software must accomplish.
- Performance requirements specify measurable values for performance variables such as response time, throughput rate.
- Interface requirements define the software, hardware, or communication interfaces. Software interface requirements define the surrounding software environments, while hardware interfaces define the hardware configuration of devices. Communication interface requirements specify the nature of communication with the hardware and software configurations.
- Portability requirements describe possible software and hardware configurations in which the system may be ported to.

- Functions level: the level at which the function is defined should be consistent with the level of detail of the operations performed by the function. Therefore functions with operations at different levels of detail should not appear at the same logical model level.
- Coupling: the interfaces (data flows and control flows) between functions should be minimized as much as possible. This facilitates the design of weakly coupled software modules during architectural design. Weak coupling is another design quality measure which will also be described further in later chapters.
- Functional Decomposition: The number of functions defined at each level should not be more than seven. This reduces the complexity of the model at each level.
- Implementation information: any information related to implementation such as a file or a record in a file, a data structure type must be omitted from the model. It is important to keep the model at the logical level, i.e., at the level of requirements specification in which “what” the software should do, is the only focus.
- Performance Attributes: the performance attribute of each function such as timing, throughput or capacity should be clearly stated.
- Criticality: functions which are critical to the mission of the system should be clearly identified. This rule might require a criticality analysis activity to be performed. It is important to identify the critical functions which must be thoroughly verified during design and tested during implementation.

b) Specifying the software requirements

c) Analyzing the software requirements

A summary of these activities is given in the following paragraphs.

A) Construction of the logical model

The standard mandates that the developer must construct an implementation-independent model termed as the logical model of the software system. A software requirement analysis method (e.g., structured analysis) must be adopted and used throughout the SR phase. Verification techniques such as walkthroughs and inspections are strongly recommended by the standard for each level of detail in the logical model. This is to verify the completeness and correctness of the model at the current level before proceeding to the lower level of detail.

Several rules to define the logical model were strongly recommended in the standard. These rules appear to be oriented towards functional or structured analysis. They can also be used with object-oriented analysis, as the functions or methods specified for a given object, can still be defined hierarchically. These ideas will be described in detail in later chapters. The rules for defining the logical models are summarized as follows:

- Functions' Names: the names of functions in the logical model should have a declarative structure and a single specific purpose. This helps in defining strongly cohesive modules during the architectural design phase. Strong cohesion among the modules, is an important design quality measure. It will be described further in later chapters, when discussing the design phase in detail.

g) Ambiguous requirements should be addressed. All ambiguous terms should be qualified or made more specific.

h) The output documents produced in the UR phase are the user requirements document (URD), the project management and configuration management plans for the SR phase, the verification and validation plan, and the quality assurance plan. These plans will be discussed later in this section. The URD must contain the following information:

* A general description of the user expectations regarding the functionality of the software.
* A list of all known user requirements specified as clearly as possible
* Specification of all solution constraints imposed by the user
* Description of the external interfaces to the software system.

The User Requirements Review is formally conducted at the end of the UR phase. User requirements which are determined to be infeasible must be clearly tagged in the URD.

The Software Requirements Definition Phase

The main objective of the SR phase is to analyze the user requirements (or the URD) and produce the software requirements specification document (SRD). Although the SRD is the responsibility of the developer, the standard recommends that the project management should ensure that users, hardware engineers, and operations personnel are consulted to minimize the project risk factor.

The major activities performed in the SR phase are listed below:

a) Construction of the logical model

accuracy attributes. Capability requirements are often described as a sequence of operations. Constraint requirements on the other hand are those requirements placed by the user on how an objective is to be achieved. They place constraints on how the software can be built and operated. Examples include the requirements to use certain software, hardware, operating systems, protocols, or library modules. Constraints requirements on the software may include security or portability requirements.

- Requirements attributes: the following attributes must be attached to each requirement.

  a) Each requirement shall include an “identifier” (e.g. a number) which can be used for traceability to specific artifacts in subsequent phases.

  b) Essential requirements shall be clearly marked to distinguish it from those which are negotiable.

  c) Each requirement shall be verifiable, i.e., it is possible to check that it has been addressed in the design, implemented in the software, and tested for, during the testing phase.

  d) A level of priority shall be attached to each requirement to facilitate the planning of the release or production schedule.

  e) The source of each requirement shall be stated by referencing a specific document.

  f) Unstable requirements should be clearly marked as such. These requirements are subject to change during the software life cycle.

A) Capturing the user requirements

The user requirements originate first from the user needs, and should be refined and clarified through interviews and surveys. Definition of the user requirements must be an iterative process. The knowledge and experience of the potential developer should be used to collect recommendations on implementation feasibility. The feasibility is based on existing software and prototypes, and perhaps through building new prototypes.

B) Specifying the operational environment

The operational environment consists of all the external systems and the interfaces with these systems. It can be specified using a narrative description supported by block diagrams. These diagrams show the role of the software in a larger system. The context diagrams specify the external interfaces of the software. Information on external interfaces may be documented in a special document called the Interface control Document (ICD). If the external systems already exist then the interfaces and information exchange can be specified in detail and constrain the design. If this is not the case, the details of the external interfaces can be developed during the SR and AD phases.

C) Specifying the user requirements

In this activity the user requirements are classified, and attributes are attached to each requirement. These steps are described as follows:

- Classification of user requirements: user requirements are classified as either capability requirements or constraint requirements. Capability requirements are those requirements needed by the user “to achieve an objective”. They describe functions and operations needed and include performance and

The third model follows the evolutionary development approach in which all phases of the life cycle are performed to produce a release. The project plan in this case specifies the development of multiple releases. Each release incorporates the experience of earlier ones. One major reason for such an approach is that the user requirements are incomplete to start with. The standard requires that the developer should follow the user priorities and deliver the parts of the software that are both important and possible to develop with minimal technical problems or delays. One major disadvantage of this approach is that the initial software architectural design may not be easily adaptable to accommodate the changes necessary for later evolution.

2.3.1.1 ESA Development Phases

Next we discuss the details of the activities and the deliverables for each phase.

The User Requirements Definition Phase

The standard mandates that the definition of the user requirements shall be the responsibility of the user. It strongly recommends however that the expertise of the engineers and the operations personal should be used to help define and review the user requirements. There are three major activities in the UR phase,

a) Capturing the user requirements

b) Specifying the user requirements

c) Specifying the operational environment

The details of which are as follows:

- Final Acceptance of the software and the delivery of the Project History Document(PHD).

The above milestones were mentioned as the minimum set necessary for every project. It was stated that additional milestones should be specified for large projects to measure the progress of deliverables.

The Life-Cycle Models

Next, the software life cycle models are discussed. The standard does not require a particular life cycle model. It does however state that a model should be defined for each project to cover the above mentioned phases. Three models are discussed in the document prepared by the ESA BSSC. The models are presented as common approaches for the life-cycle model. The first model follows the simple waterfall model discussed in the previous section using the above phases. In this case the phases are sequentially executed with iterations allowed within a part of a phase for error corrections.

The second model specified is the incremental delivery approach in which the software is delivered in multiple releases, each with increased functionality and capability. In this case following the AD phase the DD, TR, and OM phases are executed for each release, i.e., DD1, TR1, and OM1 are performed for release 1, and DD2, TR2, and OM2 are performed for release 2 and so on. This approach is necessary for large projects since a single delivery would not be practical. However, it is necessary that each release be usable and provide a subset of the required capability. The disadvantage of this model is the increased amount of testing required to confirm that existing capabilities of the software are not impaired by any new release.

phases. Software Quality tests are also performed. The code, DDD, and SUM documents are reviewed in the formal Detailed Design Review (DD/R) by software engineers and the management concerned.

The TR phase includes the installation and the provisional acceptance testing activities. The objective of these tests is to establish that the software fulfils the requirements specified in the URD. A Software Transfer document (STD) is produced which contains the description of the activities performed and the transfer of the software to the operation team. In the OM phase, the software is monitored for enough time to establish the final acceptance testing which confirms that it meets all the requirements specified in the URD. Code maintenance activities are carried out. A Project History document (PHD) containing the important managerial information accumulated during the project, is produced during the OM phase. The PHD is then updated with information gathered in the OM phase at the end of the project life cycle.

The Project Milestones

There are six milestones mentioned in the standard, these are given as follows:

- Approval of the User Requirements Document (URD).
- Approval of the Software Requirements Document (SRD).
- Approval of the Architectural Design Document (ADD).
- Approval of the Detailed Design Document (DDD), the Software User Manual (SUM), the code, and the provisional acceptance testing readiness statement.
- Provisional acceptance of the software and the delivery of the Software Transfer Document (STD).

developer produces a project management plan outlining the whole project and containing a cost estimate for the project.

The SR phase is the software requirements analysis and specification phase. A logical model of the system is produced and used to analyze the completeness, consistency, and testability of the requirements. Building prototypes in this phase could be necessary to analyze the dynamic behavior of complex real-time systems and clarify the software requirements. A Software Requirement Document (SRD) is produced to capture the software requirements specification. The SRD is formally reviewed (SR/R) by the users, software engineers, hardware engineers, and managers concerned. The project management plan is also updated and a detailed plan for the AD phase is produced.

The AD phase deals with the construction of what is termed as “the physical model” of the software. It defines the architecture or structure of the software. The software components as well as the data flow and control flow between them are defined. Several alternative designs could be proposed, one of which is selected for further refinement. Technically difficult or critical parts of the design are identified, and prototypes can be built to analyze the validity of the design assumptions. The deliverable produced in this phase is the Architectural Design Document (ADD). The ADD is again formally reviewed (AD/R) by the same teams mentioned above. The project management plan is again updated and a plan for the detailed design phase is produced.

The activities of the DD phase include module design, coding, unit testing, integration testing and system testing. A Detailed Design Document (DDD) and the Software User Manual (SUM) are produced concurrently with coding and testing. Unit, integration, and system testing is performed according to verification plans established in the SR and AD

procedures used to manage the software development project. The product development standards will be discussed first in detail, followed by a brief description of the procedure standards.

ESA Product Development Standards

In this section ESA standards for the life cycle and its individual phases and development activities, deliverables, and milestones are described.

The Development Phases

The ESA standard mandates that all software projects shall have a life cycle approach consisting of the following basic phases:

- User Requirements (UR) definition
- Software Requirements (SR) specification
- Architectural Design (AD) specification
- Detailed Design (DD) and production of the code
- Transfer (TR) of software to operation, and
- Operations and Maintenance (OM).

The UR phase is the problem definition phase in which the scope of the software is clearly specified by the users in cooperation with the developer's teams of software engineers, hardware engineers, and managers. In the UR phase, the operational environment of the software is determined. The users requirements are captured and documented in a User Requirements Documents (URD). The review of the URD (UR/R) is performed by the same teams who have already worked on software specifications in the UR phase. The approved URD serves as an input to the SR phase. Before the UR/R is complete the

2.3 Software Development standards

Software development standards provide the means for establishing, evaluating, and maintaining quality in software products. They describe the mandatory practices as well as recommended practices through well defined guidelines for software engineering projects. These standards do not provide recommendations or make use of any particular software engineering methodology, technique or tool. They describe the tasks and activities and allow each project to tailor these activities to the application at hand and to decide the adequate methods and tools to implement them.

In this section two recent and important standards for software development are described. The software engineering standard (PSS-05-0) of the European Space Agency (ESA) [MAZZA 94] is discussed in section 2.3.1, while section 2.3.2 describes the MIL-STD-498 standard for software development [STD 94].

2.3.1 The European Space Agency Software Engineering Standard

The ESA's PSS-05-0 software engineering standard has been used by the European Space industry successfully for several years. This standard has been developed and presented in [Mazza 94] by the ESA Board for Software Standardization and Control (BSSC). The BSSC maintains the standard documents and the experience of hundreds of software engineers. The first version of this standard was issued in 1984. The second version issued in 1992, is summarized in this section.

The ESA standard consists of two parts, namely, the *product standards* part and the *procedure standards* part. The product standards part contains standards, recommendations and guidelines concerning the software product, while the procedure standards part describes the

and [PRES93], verification involves certifying that a configuration item produced in one phase is developed according to its specifications given at the start of that phase.

The term “validation” refers to the activities needed to insure that the end product conforms to the user requirements defined at the beginning of the development process.

The term “quality assurance” refers to the activity of monitoring the development, verifications, and validation tasks to establish the adherence to the specified development standard as well as the verification and validation standard. The discussion on verification and validation standards will be delayed until Chapter 6. This is being done to insure the coverage of the basic phases of the development process in Chapters 3,4, and 5. Two development standards will be discussed in detail in the next section.

Configuration status accounting

Configuration accounting involves both recording and reporting needed status information. In this activity, the accounting information needed for configuration management is assembled. Status information on configuration items, the proposed changes, and the status of approved changes are recorded and reported. A log is kept for recording the sequence of events in the development of each configuration item and change request. Reports are produced to inform user personnel or management personnel on the status of a configuration item.

An important task in this activity is the preparation of version description document for each configuration item. This document must accompany the release of each version of the configuration item. It describes the various versions of the configuration item and their applications, limitations, adaptation strategy, and their installation instructions. An example of the above is the release of a new version of the user interface configuration item. The document released with this item describes the changes in the version as well as limitations, adaptation information, and installation instructions.

2.2.4 Software Verification and Validation and Software Quality Assurance

This section introduces the basic definitions and concepts of software verification, validation, and quality assurance. These processes include activities and tasks which are accomplished throughout the product life-cycle.

The term “verification” in software development involves the activities of reviewing, inspecting, testing, checking, auditing, and documenting whether or not a software configuration item conforms to the specification. In the narrow definition given in [SOM94]

- Specify a change request (CR) originating from the customers or developers. A document is produced describing the CR and submitted to a change control board,
- A change action report that documents the change control board's action concerning a given CR. This report can also include necessary studies used by the board in formulating their action,
- A discrepancy report specifying a discrepancy between configuration items. Attributes are attached to each discrepancy to specify its criticality, priority for change, and proposed fix or corrective action.
- A discrepancy action report documenting the analysis of the discrepancy, and the action to be taken in terms of specific corrective instructions, which also specified whether the discrepancy leads to change requests.
- A status report documenting the status information on the change requests. This report provides the change control board with the status of approved change requests.

Corporate polices provide guidelines for specifying development baselines, for documenting configuration items, and for defining the role and the structure of the change control board.

Operating procedures documents the chain of activities through which the above corporate polices are implemented. They must cover all aspects of configuration management procedures.

such as operations guides, and manuals for the product are also identified as configuration items.

Configuration items must be grouped in what is termed as baselines. A baseline is identified with a project milestone in the software development life cycle. It establishes the identification of the overall software configuration at that particular milestone of the development cycle. Baselines allow for controlled and manageable changes during development as the configuration items evolve from one phase to the next or from one milestone to the next. A configuration baseline is specified by identifying the configuration items it includes. Examples of baselines are functional baseline and design baseline consisting of configuration items produced at the requirements specification and design phases, respectively. Identifying the configuration items in each baseline is the important activity in configuration identification.

Configuration change control

The activities included in this task are targeted towards approval and implementation of requested changes to configuration items. Several documents are prepared to provide the administrative policies for evaluating and approving proposed changes. These documents include change and control documents, corporate polices, operating procedures, and standards and guidelines.

The change and control documents are used to specify the following information:

- Classify changes according to their impact on the various approved baselines. A document is produced containing a change classification index depending on the change impact,

product changes procedures must be in line with the configuration management plan. The project change procedures must be coordinated with the relevant elements of the project management plan as well as the configuration management plan.

The importance of a configuration management process can be clearly understood when causes of serious problems in the projects without an adequate process, are studied. These problems usually stem from:

- changes that were unidentifiable and hence difficult to track,
- changes that are difficult or costly to remove,
- changes that were not properly documented,
- changes that were applied to the wrong versions of the software,
- unauthorized changes that may result in loss of project control, or
- changes that were producing conflicts during implementation.

The Configuration Management Process Activities

The main activities in this process are summarized as follows. For a detailed coverage of these activities the reader is urged to consult [AYER92].

Configuration identification

In this task it is important to uniquely identify each configuration item. Configuration items may include items which are used to document the results of the analysis, design, implementation, and testing phases of development. In addition, hardware specifications (e.g., processors and input/output systems, and storage devices), and software specifications (e.g., operating systems and database managements systems) for the project are also considered as configuration items. Operations and Maintenance documentations

time schedule for new projects. The level of detail specified in the schedule depends on the level of detail of the WBS and the data available from previous projects.

The project plan also contains sections detailing the tasks of project monitoring and control, configuration management, verifications and validation, and quality assurance provisions. Verification and validation as well as quality assurance tasks will be discussed in Chapter 6. The configuration management process is discussed in the following section.

2.2.3 Software Configuration Management Process

The configuration management process is the process of organizing the project deliverables into software configuration items. It then provides procedures for controlling, releasing, changing, and recording as well as reporting the status of these deliverables or configuration items. This process facilitates the essential task of administrative tracking and controlling of changes to software configuration items.

A software configuration item is an aggregate of deliverable items produced in a particular phase during the software development process. Examples of such items are software requirements specification document, Prototyping models, documented code, test plans and procedures, etc.

The identification of configuration items as well as specifying and controlling needed changes in these items during development is the major concern of this process. The task of tracking and controlling such changes and their impact on other configuration items, requires a well defined process for configuration management.

The software management process described in the previous section must establish the procedures for two types of changes. These are product changes and project changes. The

model estimates the development effort as a function of the software size in terms of the number of lines of code. The model is defined by the following two equations:

$$Ef = a * KLOC * \exp(b), \text{ and}$$

$$Dt = c * Ef \exp(d)$$

where Ef is the effort needed in person-months, KLOC is the program size in thousands of lines of code, and Dt is the development time in chronological months. The parameters a, b, c, and d are dependent on the project type. The model estimates values for these parameters based on experience in previous projects.

The second model specified in COCOMO is called the intermediate model. It multiplies *Ef* by an effort adjustment factor. This factor is estimated based on a set of product attributes, hardware attributes, personnel attributes, and project attributes. The advanced model in COCOMO builds on the previous two models by incorporating an effort adjustment factor for each phase of the software development process.

For more details on resource estimation models and on COCOMO, the reader is urged to consult Boehm's work [BOE81], Basili's work for a description of the various classes of resource estimation models [BAS80], and also Pressman [PRES93] for a simple example.

Project Schedule

The project schedule is developed using estimates of needed resources as well as the estimated development time. This is done by allocating the resources to various development phases. The WBS is utilized to develop the details of the schedule as well. Data from the time schedules of previous projects can also help in developing a detailed

proposed. In this method different measures are used to estimate complexity of individual components. These measures are given below:

- number of inputs,
- number of outputs,
- number of database inquiries,
- number of master files needed, and
- the number of interfaces in each component or module.

Such estimates can be obtained from the initial functional requirements. These initial sizing estimates can also be converted to LOC estimates.

Contingencies are applied to the obtained estimates based on experience in previous projects. These contingencies are used to inflate the estimates based on the experience that previous estimates tend to always underestimate the project size. Humphery gave a rule of thumb that estimates obtained during the requirements phase should have a contingency of 100-200% code growth. These contingency factors, however, should be supported by an accumulated historical data developed by the developing organization.

Resource estimates

Based on component or module size estimates, developed in the previous steps, a resource estimate is obtained using known productivity factors. The resource estimates are usually based on empirically derived equations that predict the development effort in terms of person-months, and the project duration in terms of chronological months.

Boehm [BOE81] introduced what he termed as COnstrictive COst MOdel (COCOMO). Three types of resource estimation models were introduced in COCOMO. The basic

In his book on managing the software process, Watts Humphery [HUM89] described the project plan in terms of five sections as given below:

Goals and Objectives

The goals and objectives of the project are usually defined during the requirements phase. This section specifies the established goals and objectives from the initial requirements and also defines the criteria for measuring the project success.

Work Breakdown Structure (WBS)

This section specifies the decomposition of large development activities into small tasks in a structured manner. These tasks are defined in the WBS according to the software. The development tasks are identified with specific components or modules. The above implies that an initial conceptual design of the product must be developed as a basis for planning. The WBS has a significant impact on estimating (in person-months) the human effort required to perform the development tasks, and develop estimates of the overall cost and time schedule of the project.

Product Size estimates

An inaccurate product size estimate will be reflected in a poorly estimated resource plan and a severely understaffed project. This section deals with estimating the size of the software components and modules identified in the product structure already defined in the previous step. The size of the software components is usually available as number of lines of code (LOC).

Since it is difficult to accurately estimate the LOC for components and modules from high-level requirements, a sizing estimation method called function point has been

Controlling the project activities

This process involves solving the day-to-day problems which arise during the project such as reallocating resources when unexpected events occur. These can be delays related to the customer staff, or non-delivery of some needed equipments, tools, or documents.

Innovating new ideas

Continuous process improvement is an essential part of the management activities. Such improvements may require new methods or tools which can either make the development process or the management process more effective and efficient.

Representing the project

An important management activity is to represent the project when interacting with the customer, equipment or tool suppliers, or with sub-contractors. The management need also to represent the project with the higher management in their organization. This is done in order to give project progress briefings on the current project risks, and to ask for more resources for the project if the need arises.

The above seven activities are not meant to be independent. On the contrary there are many overlaps between them. The planning activity is the most difficult activity in the project. It requires decomposing the requirement and producing a Work Breakdown Structure (WBS). This in turn requires the decomposition of the system into subsystems and components. The major tasks of estimating the project size, estimating needed resources, and developing a time schedule, all depend on the previous step, where the WBS is defined. The rest of the section will be devoted to the discussion about the project plan.

The project Plan

Staffing the project

This is the process of filling the staff positions identified in the previous activity with the properly skilled personnel. It also involves training and promotion activities applied to the right people who will be assigned to the project tasks.

Monitoring the progress of the project

This is the activity where the project progress is monitored to check if the objectives established in the planning process are being met. To effectively monitor the project, the management must identify a set of measures related to the objectives of the project. Milestone is the tracking measure used. A milestone is a significant event such as the end of a design or code walkthrough or drafting an important development document. This measure is mainly used to track the progress of an activity in the project.

Project milestones can be either internal (used by the development team) or external (used by the customer). As described later in the section 2.3, external milestones are described by major development events such as the production of the software requirements document, or the detailed design document, etc. Internal milestones, on the other hand, are much more frequent than the external ones.

One more example of a measured monitor can be the actual cost or level of effort spent in the project activities and how it relates to the estimated level of effort and the objectives of the project.

focus and the important resource. However the current technology of tools is far from being in an automation state as manufacturing systems are currently based on robots.

The main focus of the software industry is the process-based approach using people and tools as well as metrics to allow for process improvement. This will in turn increase predictability and reduce variability. As mentioned by Watts Humphery in [HUM89]:

> "The basic principle are those of statistical process control where a process is said to be stable if its future performance is predictable within established statistical limits".

The Management Process Activities

The set of activities that characterizes the management process, are described by Ince in [INCE93]. These activities are summarized as follows:

Planning the software project

Assuming that the project goals and scope have been established, planning is the process of defining the project activities, subdividing large activities into smaller tasks, and estimating (in person-months) the human effort required to carry them out. It also involves estimating the overall cost and time schedule of the project.

Organizing the software project

This activity specifies the type of staff in terms of staff positions needed for the project tasks, the duties and responsibilities of each staff position, and the channels of communications between the staff members. Organizing also involves specifying the communication channels between the customer's staff and the developer's staff.

CASE tools for rapid prototyping which incorporate user interface tools, program generation, and expert systems designed to automate as much of the work as possible. CASE tool environments will be discussed later in this chapter.

2.2.2 Software Development Management Process

The management process of software development consists of a set of procedures and activities needed to manage the project. Throughout the life-cycle, the management process overlays the development process discussed above. An analogy can be drawn here between the production of an engineering product on a production line of a plant and the control process which controls the plant. The control process manages the production line to oversee that the development process is done smoothly and efficiently (i.e., with minimum cost), and that it produces quality products according to the set standards. Similarly the objective of the software development management process is to produce high quality software within the scheduled time and within the estimated cost.

Yeh [Yeh93] has proposed a process-based approach for software management which aims at making software production more predictable and improve process capabilities. He contrasted this approach with what he termed as the “all-star” approach and the “robotics” approach. The *all-star* approach is based on using the best experienced people or the “elite” team in software development in order to reduce risk and maximize efficiency. While having experienced staff is definitely good in any project, in large projects it is difficult or even impossible to find the number of experienced people required to do the job. Therefore, the *all star* approach is deemed impractical.

The other approach mentioned in [Yeh93], the so called *robotics* approach, is based on automation as an analogy with factory automation. Here integrated tools become the main

The evolutionary model is the basis of the spiral model (Figure 2.7) of software development presented by [Boem88]. The term “spiral” is used in order to highlight the fact that in each development cycle the software evolves until it is completed. Each cycle or iteration of the spiral consists of three major parts. These are:

- Project planning and risk analysis for the next cycle (either based on the initial requirements or based on customer reaction or experience with the previous release)
- Engineering development (using either prototyping or the standard waterfall development phases), and
- Customer and user evaluation.

The first part develops and refines the detailed schedules and the plans for the next release. Risk analysis is used to assess uncertainty in the requirements and refine the plans and schedules to include prototyping and simulations for the high risk parts of the software. Engineering development then begins with the scheduled development activities followed by customer and user evaluations. This cycle is repeated for next release with lessons learned and new or refined requirements obtained based on the last part of the previous cycle i.e. customer and user evaluation.

The main advantage of both the incremental and evolutionary models is that the user or customer evaluation is part of the development cycle. This ensures that risk due to customer dissatisfaction is minimized.

T.G. Lewis [LEWIS 91] gives an interesting discussion on the spiral model and its relations to prototyping-based development models. He maintains that repetitious development is a costly way to build software unless automated tools are used. He advocates the development of

changes in mission technology which may change the requirements are anticipated. Recall that the incremental model assumes completely defined requirements and an architectural design before the first build is produced.

Opportunity items describe the favorable conditions in which a model would be used. The final outcome from this table favors the use of the evolutionary model as specified in the decision row at the bottom of the table. The reason is that for this model there is only one medium level risk item, and more high level opportunity items as compared to other models.

The incremental model when combined with prototyping could mean that a prototype will be built incrementally with testing and user evaluation activities to evaluate the current prototype after each iteration. In this case there cannot be multiple releases and when the user's concerns are addressed, the product is finally developed.

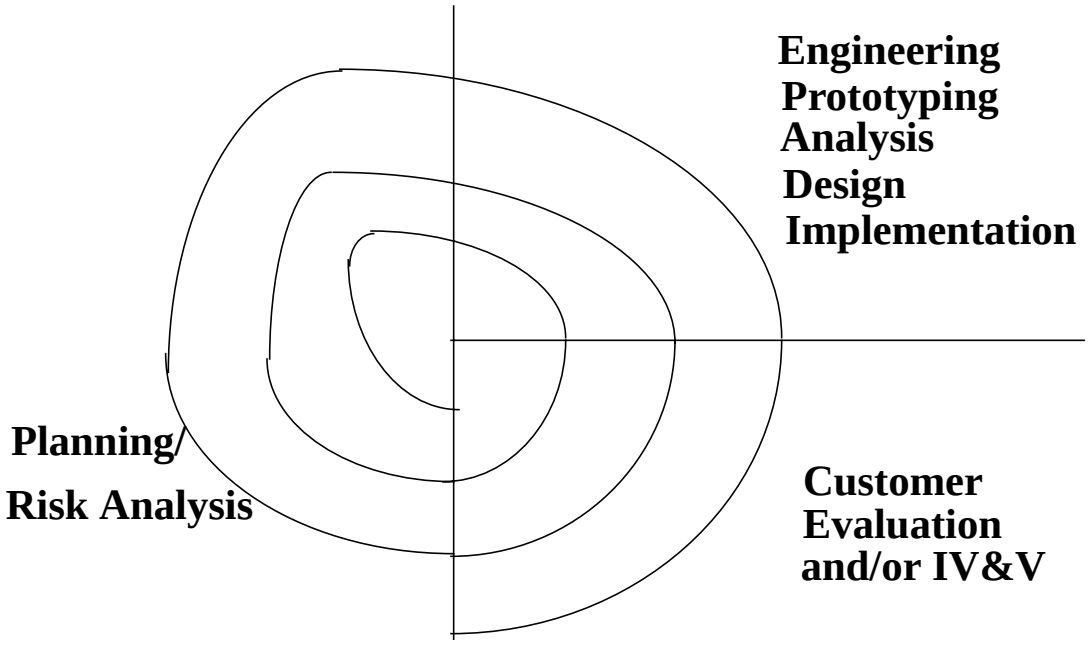


FIGURE 2.7 Spiral model

	Water Fall		Incremental		Evolutionary	
Risk-Item no.	**Risk Item (Reasons against this strategy)**	**Risk Level**	**Risk Item (Reasons against this strategy)**	**Risk Level**	**Risk Item (Reasons against this strategy)**	**Risk Level**
1	Requirements are not well understood	H	Requirement are not well understood	H	User prefer all capabilities at first delivery	M
2	System too large to do all at once	M	User prefers all capabilities at first delivery	M		
3	Rapid changes in mission technology anticipated--may change the requirements	H	Rapid changes in mission technology are expected --may change the requirements	H		
4	Limited staff or budget available now	M				
Opp-Item no.	**Opportunity Item (Reasons to use this strategy)**	**Opp. Level**	**Opportunity Item (Reasons to use this strategy)**	**Opp. Level**	**Opportunity Item (Reasons to use this strategy)**	**Opp. Level**
1	User prefers all capabilities at first delivery	M	Early capability is needed	H	Early capabilities is needed	H
2	User prefers to phase out old system all at once	L	System breaks naturally into increments	M	System breaks naturally into increments	M
3			Funding/staffing will be incremental	H	Funding/staffing will be incremental	H
4					User feedback and monitoring of technology changes is needed to understand full requirements	H
					Decision: Use this strategy	

Table 2.1: Sample risk analysis for determining the appropriate program strategy. Risks and opportunities. The “DECISION” entry on the bottom line shows which strategy was selected, (Adopted from Appendix G of MIL-STD-498.)

According to the risk-item no. 3 in Table 2.1, the user prefers all capabilities of the software at the first delivery. It is not a high risk item for the incremental and evolutionary models. The reason here is that it does not change the software requirements. On the other hand, it is considered as a high risk item for both the waterfall model and the incremental model if rapid

items (positives) for each model. Each item is assigned a risk or opportunity level of High, Medium, or Low. Then a decision is made about the strategy to be used, based on a trade-off among the models like:

- Waterfall model
- Incremental model
- Evolutionary model

The risk-item no. 1 in Table 2.1, states that the requirements are not well understood (which is true with many projects). This is a high risk item for both the waterfall model and the incremental model. This item is a major source of costly errors in the water fall model as mentioned before since all phases of the development will have to be redone.

For the incremental model risk-item no. 1, still is a high risk because the development phases for the first release will have to be revised if requirements are not well understood. The evolutionary model, however, assumes that the requirements are only partially understood and develops the first release on that basis.

The risk-item no. 2, in Table 2.1 is only of medium level risk for the waterfall model since the developer may have the experience and capability to develop large scale software-systems. This condition does not exist in the other two models since the development there is divided into several releases where the first release will only have a part of the software functionality.

necessary for later evolution. In this case much more time may be spent in redesigning the software architecture in subsequent releases.

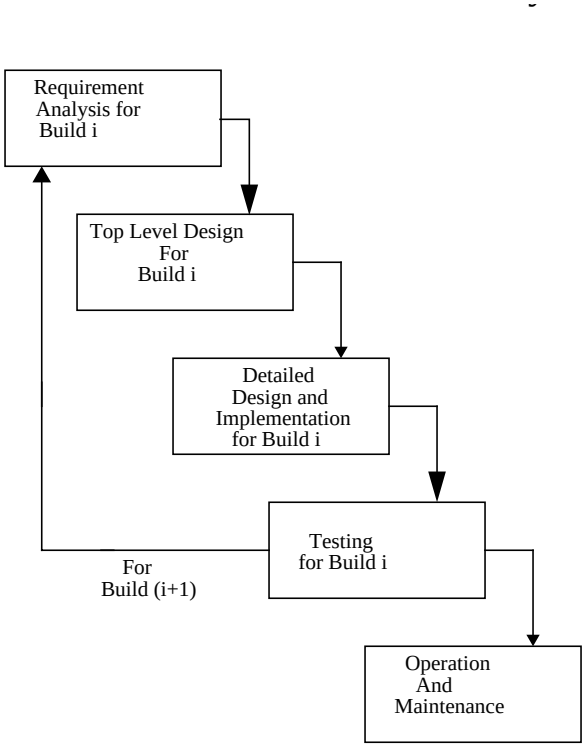


FIGURE 2.6 Evolutionary Development Model

The incremental approach learns from the evaluation and testing of the previous build to improve the quality and functionality of the product in the subsequent builds. With this approach, it is assumed that the user requirements are completely defined before the development of the first build can start. The evolutionary approach, on the other hand, assumes that the user requirements are only partially specified and it evolves with the user experiences with each release.

Table 2.1 illustrates a risk analysis approach for selecting an appropriate development process model. The approach consists of listing conditions for risk items (negatives) and opportunity

preference. The customer may prefer to have all the capabilities of the software in the first delivery.

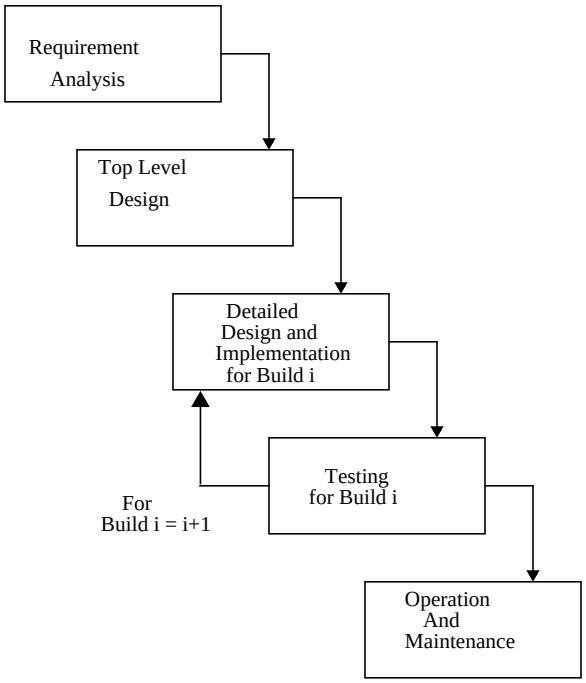


FIGURE 2.5 Incremental Development Model

The second model as shown in Figure 2.6, follows the evolutionary development approach in which all phases of the life cycle are performed to produce a release. The project plan in this case specifies the development of multiple releases. Each release incorporates the experience of earlier ones. One major reason for such an approach is that the user requirements are incomplete to start with. The standard approach is that the developer should follow the user priorities and deliver the parts of the software that are both important and possible to develop with minimal technical problems or delays. One major disadvantage of this approach is that the initial software architectural design may not be easily adaptable to bear the changes

- An understanding of the way the process (whether it is the development process, the management process, the configuration management process, etc.) can be used to control the risk.
- A questioning or a critical thinking approach should be embodied in the building of the project plans.
- Emphasis should be put on obtaining quantitative measures of important characteristics in order to control it.

The type of development process model which allows for a greater ability to control risk is called a risk analysis based process model. The development phases in this model are scheduled to reduce the project risk. For example the prototyping based model described above can, in a way, be called a risk analysis based development model. This is due to the fact that a prototyping step is used to aid in the process of defining the user requirements or to assess in discovering the potential design and implementation problems.

In order to decrease the project risk factor, two models have been suggested in practice. The first model is based on an incremental development approach, as shown in Figure 2.5. In the incremental model, the software is developed in multiple builds (or versions), each with increased functionality and capability. In this model, following the architectural design (or high-level design) phase, the rest of the phases in the waterfall model are executed for each build. This approach is necessary for large projects since a single build may not be practical. However, it is necessary that each build should be usable and provide a subset of the required capability. The disadvantage of this model is the increased amount of testing required to confirm that existing capabilities of the software are not impaired by any new build. The builds produced in each cycle may or may not be released depending on the customer

Models based on risk analysis

The term risk factor has an implied definition of risk as a measure of uncertainty in achieving the project goals such as developing a product which satisfies users needs while meeting the project deadlines. Some of the important factors affecting the overall project risk factor are based on uncertainty in:

- understanding users needs,
- assessing the difficult technical problems which might show up during design and implementation,
- handling changing requirements which arise due to technological advances or changes in the user needs.

Risk analysis is one of the important and essential management activities (the management process is described in the following section). The focus is on assessing or getting quantified measures for the above mentioned uncertainty factors which have common roots in many projects. Risk management is the process of managing the project risk throughout the development phases. It consists of the following activities:

- Risk identification,
- Risk assessment,
- Risk prioritization,
- Risk management strategies,
- Risk resolution, and
- Risk monitoring.

Successful risk management is based on three basic patterns of thinking as identified in [DOWN94]. These patterns are specified as follows:

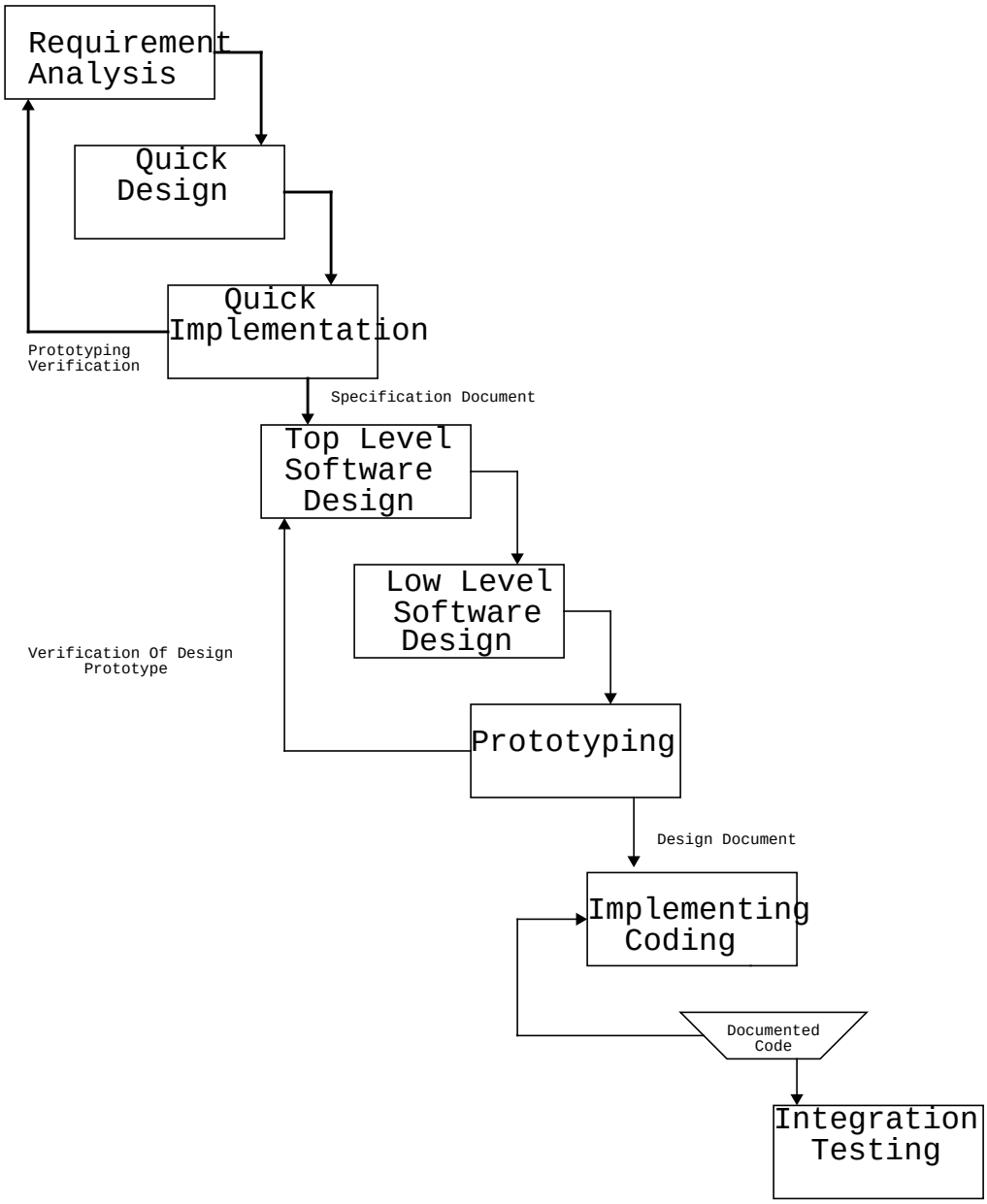


FIGURE 2.4 Prototyping based development during design phase

tools to study the dynamic behavior of the software as it reacts to external events. These models are important to study the timing behavior by simulating the external environment, as well the state transitions which take place as the software executes its functions in reaction to external events. Issues such as scheduling real time tasks in a multiprocessing environment, estimating synchronization, and resource contention delays can all be studied using such models. These models can be used to guide the analysis and design activities. Tools support for these models is discussed in the last section of this chapter and in Chapter 6.

A development process model may require the use of a prototype for developing a clear, consistent, and complete specification. This activity is based on getting feedback from users and other software developers. Several iterations on refining and presenting the prototype may take place before the specifications are finalized. The process would then proceed in a sequential fashion following the development phases in the waterfall model as shown in Figure 2.3.

Prototyping may also be used during design and implementation. In this case dynamic simulation models are used to guide the development activities and get feedback from users and other developers on the current status of the evolving product as shown in Figure 2.4.

Next we will discuss a type of process model which builds on the above and uses incremental or evolutionary development ideas to reduce the project risk factor in developing a high quality software product.

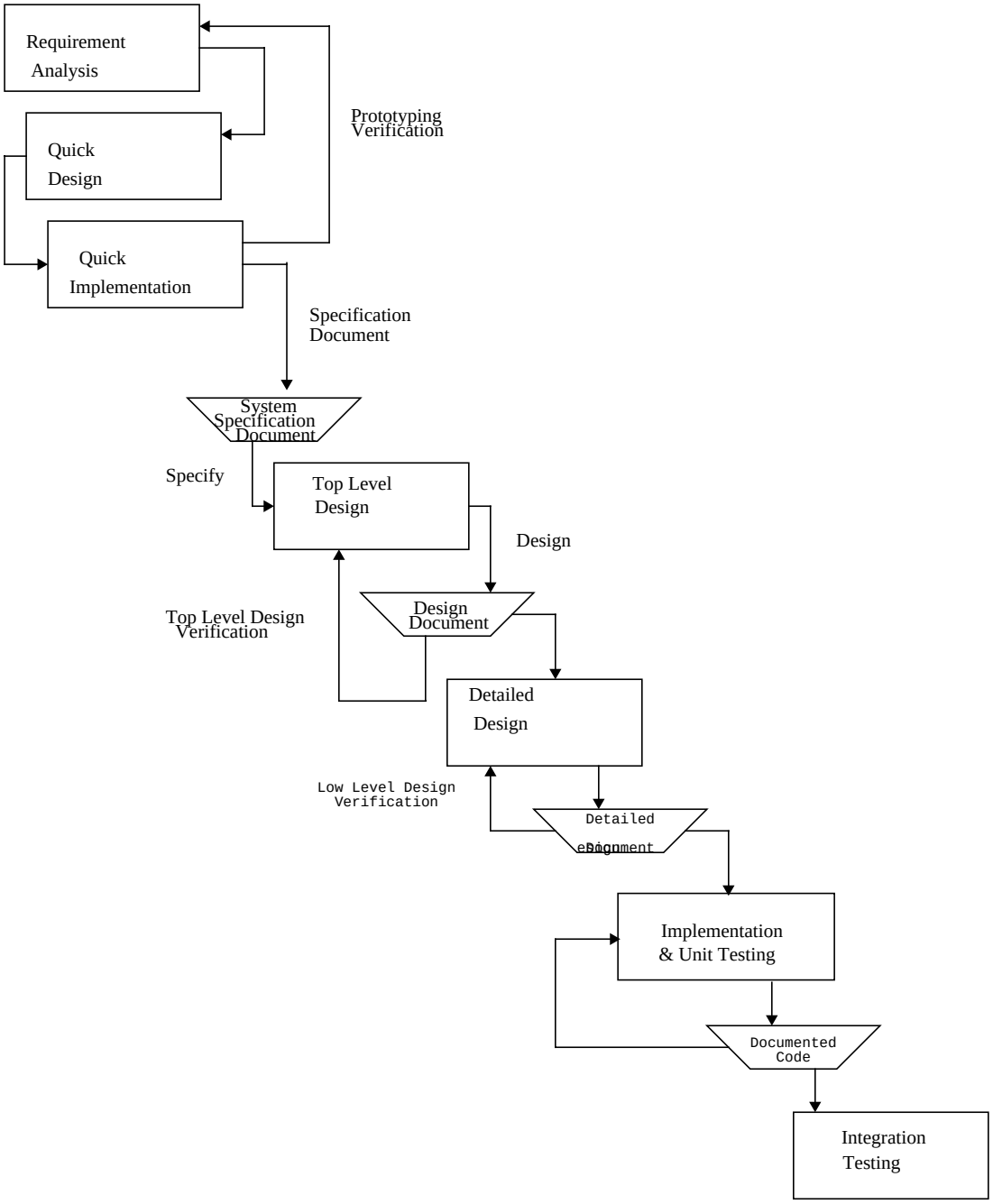


FIGURE 2.3 Prototyping based development

The broad definition of prototypes used in this text includes simulation models and executable specifications. These are dynamic models developed using special modeling and simulation

accomplish. The whole development process must be augmented with needed activities such as prototyping. It needs to be re-structured around the iterative development concept rather than sequential development. These ideas are discussed next in the context of defining a more realistic development process.

Prototyping Based Models

The classical definition of prototyping implies the development of a “working version” of the product or parts of the product with functionality covering usually the difficult part of the functional requirements for the product. Difficult requirements refer to those that are vague.

Vague requirements are not defined precisely. They are poorly understood, or could be dependent on strict timing, and safety constraints. A working version may be developed through quick analysis, design, and implementation following similar activities as specified in the waterfall model but executed in a much faster pace to come up with a quick, dirty version of the product. The “dirty version of the product” refers to a version with imperfections in screen layouts, incomplete validation of input data, inefficient use of the storage resources etc. Moreover the time consuming detailed documentation process is skipped since an executable version of the product to be developed is obtained rather than writing specifications in words.

the design (traceability). Design reviews are carried out to look for logic faults, interface faults, lack of exception handling, and most importantly non-conformance to the specifications or lack of traceability.

During the implementation phase, a documented code is obtained from the detailed design document. Unit testing is then carried out on each module to verify that the module correctly implements its detailed design.

Integration testing is then carried out when the modules are integrated and tested to determine if the system, as a whole, functions correctly.

The waterfall model emphasizes the need of complete requirement specifications before doing design, and the need of complete design specifications before the implementation phase. This process model is heavily document driven. The requirements specification documents and the design documents must be written in great detail. These documents are formally and rigorously reviewed before proceeding to the next phase. The main goal is to detect and remove defects before they propagate to the next phase. The main advantage of the waterfall model is to facilitate the implementation of the management activities of estimating the cost and schedules of the development process. This simple sequential development process is also easier to track and tailor to large projects.

According to many experts in the field, however, (see [Boehm 88], the waterfall model does not capture the realistic sequence of activities and tasks required for modern software development. The problem comes from requiring the developer to write detailed specifications of poorly understood requirements. For complex real-time systems in particular, the task of writing detailed adequate, consistent, and correct specifications based on vague, ambiguous, imprecise, contradicting, or incomplete user requirements is almost impossible to

The software specification document is not a design document. It should set out “what” the software should do without specifying “how”. During the creation of this document, errors (or inconsistencies) in the requirements’ definition statements are discovered and redefined accordingly with the software requirements definition team.

The design phase determines how the software is to meet the specifications. The two most important activities during design are decomposition and refinement.

Decomposition is the process of partitioning the configuration item under development into smaller modules. Interfaces between these modules must be precisely specified. Each interface specification provides the module’s clients with the information needed to use the module without knowledge of its implementation, at the same time it provides the implementor with the information to implement the module without the knowledge of its clients. The interface provides a place for recording design decisions.

Refinement involves working at different levels of abstraction; perhaps refining a module at one level to be a collection of modules at a lower level, hence the term architectural design is used to define this phase of the design process.

The design documents consist of two parts:

- architectural design; a description of the software as a set of components along-with their couplings or interfaces.
- detailed design; a description of the design of each component as a set of related software modules.

Design verification is then carried out for checking not only that the design is in accordance with the specification, but also that every specification statement is reflected in some part of

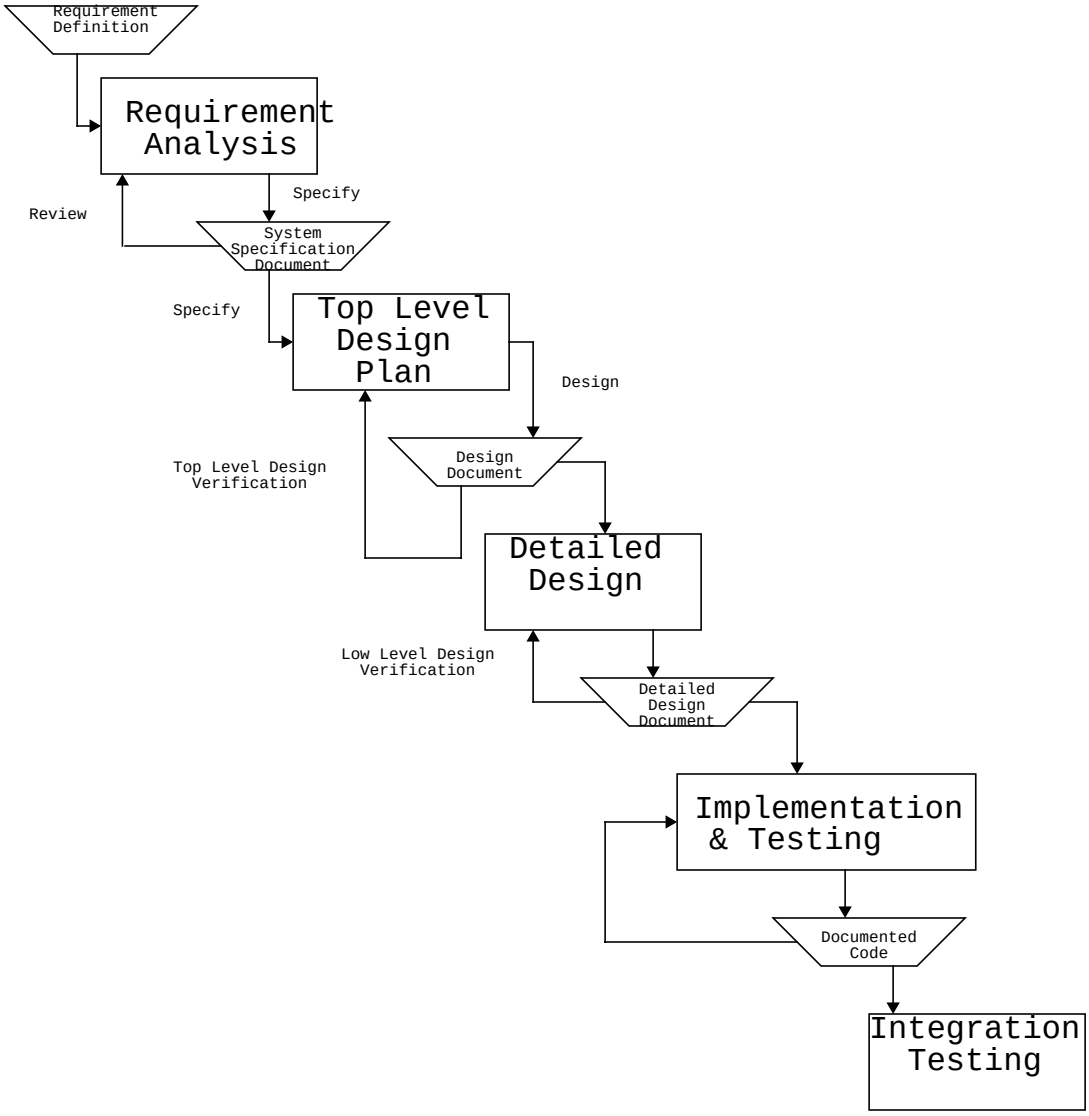


FIGURE 2.2 The waterfall model

In the requirements analysis and software specification phase a detailed and precise description of the software's functional, timing, and data requirements are developed and documented. The activities in this phase must be carried out and reviewed before the high-level design activities can start. An important outcome of this phase is the software specification document.

The Waterfall model

The waterfall model (Figure 2.2) for software development follows the classic life cycle model which assumes a sequential development process consisting of several development phases. The process iterates within each phase to correct problems found during reviews and verification activities. It also iterates back from the maintenance and operation phase to the early development phases in order to correct problems found during operation or to deal with new requirements. The development phases are explained in the following paragraphs.

Given a set of requirements for a software configuration item (specified in a requirements definition document), the software development process starts with the requirements analysis and specification phase. This is followed by a preliminary design (or high level design) phase, and then a detailed design phase. Once the design is finalized, the coding or implementation phase begins which is followed by the testing and validation phase. Verification activities are conducted throughout development in order to verify that the end products of each phase are consistent with its input documents.

Although the requirements and design activities influence each other as they develop, the waterfall model assumes a strictly sequential development. Iterations are only allowed within a phase. Therefore, mistakes found during the review process in terms of inconsistencies with the input document (i.e., the requirements definition document in this case) are documented. The activities in this phase are repeated to resolve all problems found during the review process.

2.2 Software Engineering and the Software Development Process

Software engineering refers to the concept of dealing with software as an engineering product which must be developed according to a well defined engineering standard. In fact the terminology of “the software factory” has been advocated to highlight the concept of a well defined industrial software development process [FERN 92].

Many engineering studies have been conducted to establish models for the software development process or the software life-cycle. The development cycle of real-time software, in particular, is very complex due to the many distinguishing characteristics mentioned in the previous chapter. These characteristics include concurrency, reliability, safety, and timing requirements. Such characteristics require rigorous verification and validation activities. In the following section, several types of development models are discussed. Later, in the section 2.2.2, the activities or tasks required to manage the development effort are described.

2.2.1 Software development models

Next we discuss several well known software development models. The waterfall model is discussed first. This model is the oldest development model that has been widely used in industry for several years. The waterfall model is easy to manage and the planning process is relatively straight forward. It is however inefficient in terms of the high risk factor in achieving the goals of high product quality and cost effectiveness measured by the time required to produce the product. The concepts of prototyping and risk analysis based models developed to overcome the inefficiencies of the waterfall model will also be discussed.

As shown in Figure 2.1, the full-scale development phases consist of the following basic set of activities:

- The system requirements definition and analysis phase: In this phase the system concepts are developed and a hierarchy of system requirements is produced. These requirements are then analyzed for consistency, completeness, and testability.
- The system top-level design phase: In this phase, the system configuration is designed in terms of hardware and software configuration items. These items and their interfaces are specified. Software and hardware requirements for these components are specified as well.
- The system detailed design and implementation phase: During this phase, each configuration item specified in the previous phase is developed and tested. The development process of software items will be discussed in detail in the next section which covers software engineering and the software development process.
- The system integration and testing phase: In this phase, the developed software and hardware items are integrated and tested. The system requirements are validated in this phase.

Several documents and reviews are prepared and performed during each of the above phases. A detailed specification of the above activities, necessary documentation, and reviews are explained in section 2.3.

The preliminary activities establish the system concepts or conceptual basis, then the full-scale development phases start, and finally, the production and deployment phases follow. Figure 2.1 gives an overview of the system development process.

The system concepts are established by capturing the user needs or requirements, and defining the scope of the system. This phase produces documents with titles such as “The Mission Needs Statement”, “The Operational Requirements”, or “The User Requirements Document”.

Capturing the user requirements involves the following:

- Concept exploration,
- The use of documented experiences with other systems,
- The use of specially developed prototypes, and
- Simulations.

The activity of capturing the user requirements is the most crucial activity in the development of any system. No other activity in the development process can increase the project risk factor (of not solving the user problems) to a higher level. Mistakes made in this phase are propagated to subsequent phases and it becomes very difficult to rectify these mistakes later on.

Realization of these risks has lead to the evolutionary approach of system development. In this approach the system is developed as a sequence of builds. The first build incorporates part of the needed capabilities. The next build adds more capabilities, and so on, until the system is complete. The assumption here is that every user need is not fully known or understood up front so the requirements cannot be fully defined. User needs and system requirements are partially defined at first. These requirements are then refined in each subsequent build. The evolutionary model is one of several models of the development process which will be discussed in the next sub-section.

established before the development process for the system-components can be fully defined. Therefore, software development process models must be defined within the bounds of the system wide development process. Software engineers must be involved with hardware engineers and systems engineers early on in the system development process.

The system development process is defined by a set of integrated guidelines organized into a series of phases, tasks, subtasks, and activities that make up the development cycle of a system. These guidelines characterize the development process model by describing the different phases involved. The guidelines define the end products in each phase as well as the tasks and activities needed to produce them.

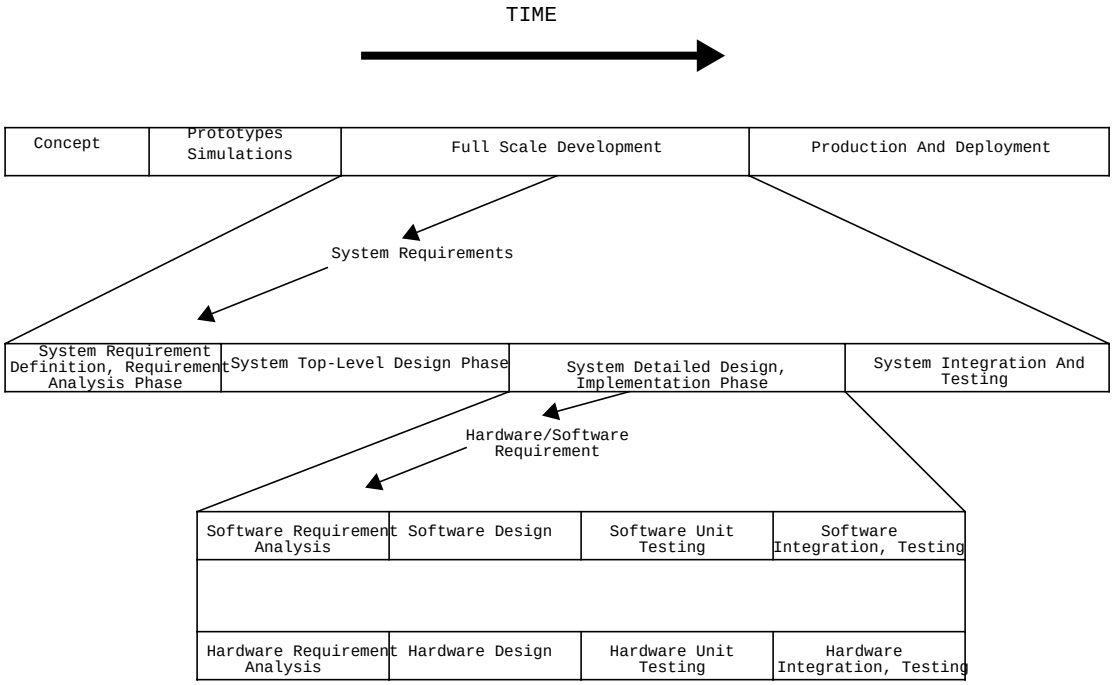


FIGURE 2.1 Overview of the System Development Process

2.1 The System Life Cycle Model and the System Development Process

In this section, the basic concepts of system development activities are described, we first give the general definition of the system life cycle model. Then the system development process is discussed.

The system life cycle model is defined as the framework containing the processes, activities, and tasks involved in the development, operation, and support of a system. This framework spans the life of the system from the definition of its requirements to the termination of its use.

The word “cycle” in the above definition refers to the way a system usually evolves through several cycles of development and enhancement during its life span. The concept of a development process model discussed next is an important part of the system life cycle model. It covers the activities and tasks starting from the definition of requirements to the deployment of the developed system.

For manufacturing engineers, a manufacturing process specifies the phases in which a product develops from raw materials to a tested, working product. The product evolves and develops as it goes from one phase to the next in a well defined and orderly manner. Flexible manufacturing systems are based on a process designed to be adaptable to the development of several different products. A complex manufacturing process is always specified by a model called the manufacturing process model.

Software products also require a well defined development process model. Software products, however, are always part of a larger system consisting of hardware components and software components. The system-wide concepts, requirements specification, and design must be

MIL-STD-498 standard. These standards are discussed along-with their tasks, required documentation, and procedures to perform reviews. The special problems and risks of real-time software development are then discussed in section 2.4. Section 2.5 presents a discussion on the past, current, and future trends of ICASE technology. In section 2.6, the integration aspects of Integrated CASE environments is discussed. Finally in the last two sections of the chapter, two examples of ICASE environments are presented in sections 2.7 and 2.8. These environments include

- Teamwork by Cayenne Software Inc. and
- Software through Pictures (StP) by IDE.

Chapter II: The Software Development Process

The objective of this chapter is to introduce the models and standards of the software development process. Also described in this chapter are the evolving ICASE technologies in support of these standards, and some examples of ICASE environments which will be used throughout this text.

Section 2.1 gives an overview of the system development process. The concept of the development cycle in terms of a phase-by-phase process model is presented. The software development process is then discussed in detail in section 2.2. Several process models are described and then contrasted with the system process model. The Software Engineering issues of Process Management and Configuration Management are also discussed in section 2.2. Section 2.3 describes the industrial standards for software development. Two widely used standards for software development are the European Space Agency standard (PSS-05-0) and the US DOD