Chapter 10

Software Engineering Processes

CONTENTS

This page intentionally left blank.

Chapter 10

Software Engineering Processes

"Because software, like all capital, is embodied knowledge, and because that knowledge is initially dispersed, tacit, latent, and incomplete in large measure, software is a social learning process ... in which knowledge that must become the software is brought together and embodied in the software." – Howard Baetjer, "Software as Capital"

10.1 Introduction

In the earliest days of computers, memory was small, language consisted of binary machine code, and programmers were adventurous souls from other disciplines who made up the craft of programming. While early programmers could get by writing some code and using clever tricks, often undocumented, those were not the "good old days." As technology advanced, so did the need to build bigger and ever more complex programs. Software development reached a point where it had to be developed by teams. There had to be requirements, plans, detailed designs, actual building, testing, development of efficient, intuitive user interfaces, and integration into larger systems of computers, machines, and people. The discipline to develop software in this manner is known as software engineering, a complex process that itself requires many sub-processes.

To put up a tent you need a level spot of ground, located in a small clearing. A tent is a relatively simple form of shelter with few requirements. It can be erected quickly, with little planning, and at little cost, but it does have its limitations. Contrast this with building a house. Much more is needed by way of planning, materials, cost, time, and work. Presumably, the builder or future owner of the house has some specific and general requirements to be met. These will need to be turned into some type of floor plan, which must then be approved by the requirement giver. There may even be artists' renditions of what the completed house will look like. Following the top-level plan, designers will put together detailed plans for each room and all the major components of the house. Again, these must be approved. When the design is satisfactory, the building will start and proceed until the house is finished. There will be inspections along the way and inspectors will examine and test various parts of the house for quality, functionality, and adherence to the plans and standards. There will probably be some changes during the actual building to make up for unknown variations in the building lot, available materials, or other factors. The future owner may even request a change after seeing what the plan is going to produce. When the structure is complete, designers or owners will decide on paint colors, carpets, lighting, and furniture. Sometime before being used, the house will need to be connected to various utilities, sidewalks, and roads. This will make it a part of the community. The primary differences between tent and house are the levels of complexity and functionality. Requirements for greater functionality result in a more complex system, which requires a more complex construction process.

Software engineering is younger than other engineering disciplines and is based on an industry that is undergoing unbelievable change. Because of its relative youth, the software industry tends to be populated by younger, creative, innovative people. All these combine to produce a constantly evolving discipline. Just as software life cycles have multiplied (see Chapter 2, Software Life Cycle.), so too has software engineering diverged into many different methods, each with its own disciples and champions. It is not possible to sample even the major methods here, but most development methods incorporate, to a greater or lesser degree, a standard set of activities common to most software engineering. These common pieces, framed in a "standard" engineering methodology will form the basis of our study.

10.2 Process Description

Software engineering goes far beyond learning how to write programs with computer languages. While there are aspects of software engineering which are used more effectively, or even exclusively, with specific languages, the discipline is not dependent upon language, notwithstanding the divine qualities some programmers may ascribe to their chosen language of worship. The following statement warns of the true position of programming in the scheme of software engineering; there is much more to software development than programming.

"The sooner you begin writing code, the longer it will take to get done."

Much of software engineering is associated with the software life cycle process employed (See Chapter 2, Software Life Cycle). The software development model used in a project is a portion of the overall project life cycle, and should fit within the project life cycle. The classic software development waterfall model is shown in Figure 10-1. [1] This model embodies all the major aspects of the software engineering process. The prototyping activity is not part of the classic model but is commonly used to provide feedback and allow iterative development where requirements or solutions need further definition. Other development models are often used. In addition to Figures 10-2 and 10-3, the life cycle models presented in Chapter 2 show some of the more common models.

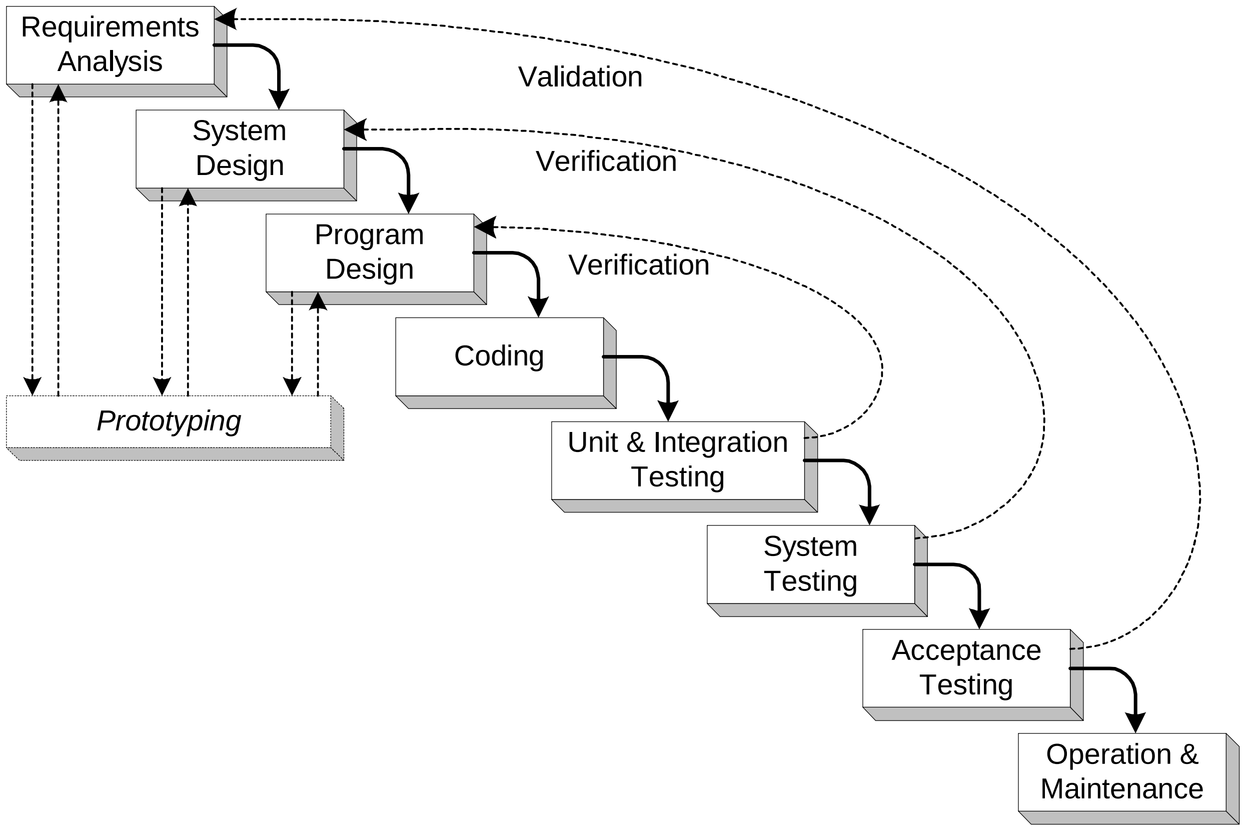

Figure 10-1 Classic Waterfall Software Development With Prototyping Modification [1]

The waterfall model has each activity flowing into the subsequent activity following its completion. While engineers look ahead and try to anticipate issues and needs that will surface later in the project, the waterfall is limited in its ability to provide feedback and allow the work of earlier steps to be modified. The prototyping modification shown in Figure 10-1 can help alleviate this deficiency somewhat.

The V model improves on the sequential nature of the waterfall model by planning and preparing acceptance tests in conjunction with the requirements analysis. System tests are established during system design and unit and integration tests are planned during program design. The tests are not performed until later in the development process, but developing tests in conjunction with the applicable requirement or design activity facilitates a more unified and focused development effort. Figure 10-2 depicts the V model.

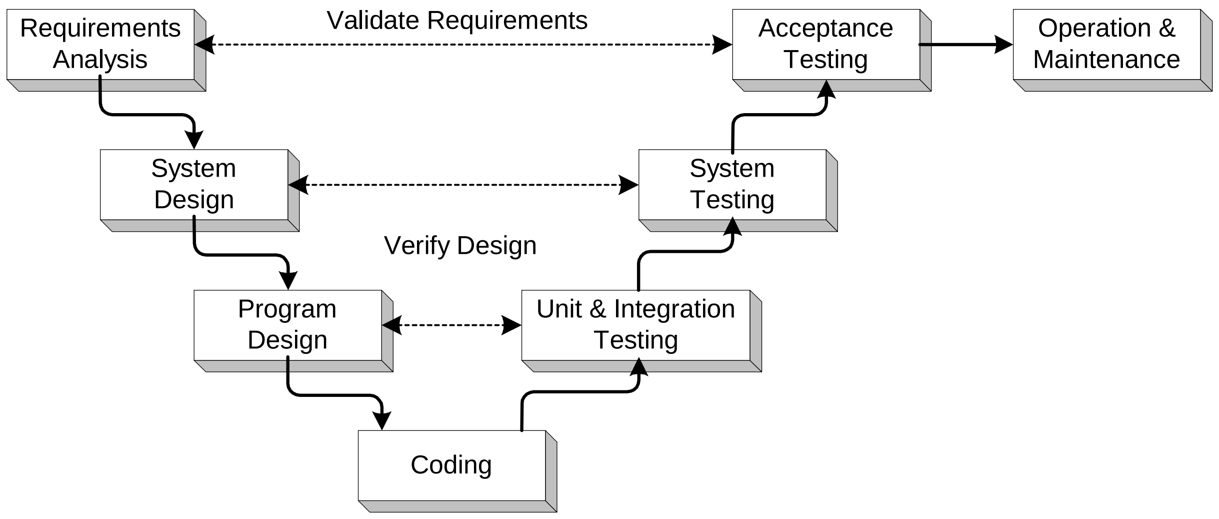

Figure 10-2 V Software Development Model [1]

The shark tooth model, shown in Figure 10-3, incorporates prototyping and involves the user in providing feedback to the development process. Users are involved in the system requirements analysis, following which, the developers perform software requirements analysis. A prototype is produced and shown to the user. Requirements are refined and development proceeds through different design phases, involving reviews and further prototype demonstrations as necessary. The prototypes allow the user and developer to better understand each other and produce a product more in line with the user's real needs.

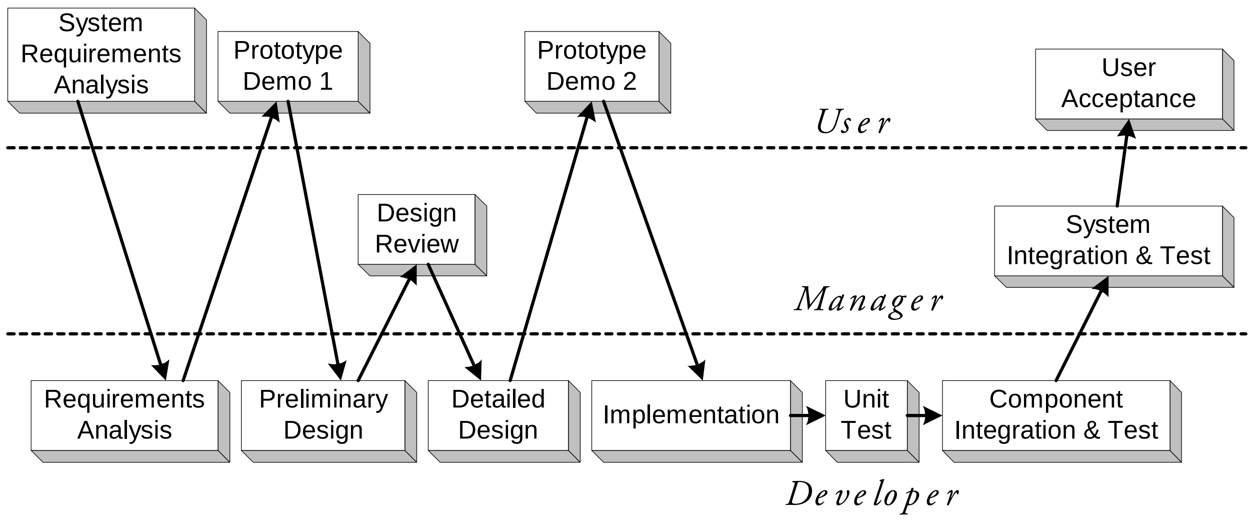

Figure 10-3 Shark Tooth Development Model [1]

The DoD requires the use of the spiral development model (see 2.2.1.4) wherever possible. This model is more iterative than the shark tooth model and at first glance appears far more complex. However, careful inspection will reveal that it employs the same software engineering activities as the other models. In other words, there are several development models, along with variations on those models, but there is a standard set of software engineering principles which can be found within the framework of most development life cycles. It is the job of the Software Engineer (SE) to apply these principles in a systematic, disciplined, and quantifiable approach to software development. [1] A partial list of the skills and knowledge areas that make up software engineering is shown below.

- Project Planning
- Requirements & Specification
- Software Architecture

- Design Methods
- Data Structures
- Algorithms
- Modeling
- Testing
- Demonstrations
- Software Components Design
- Languages, levels, constructs, syntax
- Human Factors
- Measurement & Estimation
- Verification and Validation
- Communications
- Coding (Programming)
- Operating Systems
- CASE Tools
- Software Quality
- Numerical methods
- Life Cycles

Key areas of this list are discussed in the following paragraphs.

10.2.1 Software Project Planning

As in every other discipline, planning is an essential skill for developing software. Planning includes determining what activities need to be done, in what order, when, and by whom. It also includes the following:

- What tools will be used
- How activities will be divided
- Scheduling
- Requirements management
- Quality management
- Design methods to be used
- Problem solving approach
- Demonstration methods
- Documentation, internal & deliverable
- Test methods
- Configuration control
- Design reviews
- Risk management

Responsibility for all this planning is shared by the project manager and members of the development team, including individual engineers. Each member of the team must be able to plan, or contribute to the planning of those areas with which he or she is involved.

10.2.2 Requirements and Specification

Requirements form the basis of the goals for which software is developed. Seldom are requirements known in complete detail, and engineers must elicit requirements from stakeholders to find out what is really needed and wanted. It is far more than asking questions. It involves developing and asking the right questions, perhaps educating both developers and users so they can reach a point of common understanding. It also consists of being able to communicate with users in such a way as to make sure each party knows how the other perceives the requirements.

The requirements process, summarized in Figure 10-4, has two major activities and two major products. The requirements gathered during the elicitation activity are documented in a system specification. This document is written in natural language (common everyday speech) and describes what the final system or software should do for the users. It can be general in nature or very detailed. Both stakeholders and developers should be in agreement that the system specification adequately and accurately describes the new system and its capabilities. Note that some analysis is usually required to produce the system specification.

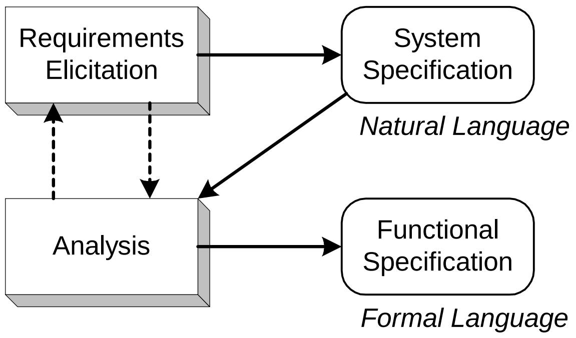

Figure 10-4 Requirements Elicitation and Analysis [1]

If it is not possible to implement the system requirements due to time, money, technology, or other constraints, the developers negotiate with the stakeholders to agree on a set of requirements that can be performed under the given constraints.

When the system specification is complete, its requirements are analyzed to determine what will be needed to comply with them. Analysis is the primary activity here, with occasional returns to the users to elicit information to clarify ambiguities. The product of the requirements analysis is a functional specification containing detailed requirements. The functional specification not only differs in its level of detail from the system specification, but also in its format. It is a formalized description of the system, presented in a language that better describes what must be accomplished in terms of software functionality. It gives specifics of what must be accomplished. It often includes relationship diagrams, data flows, use case diagrams, and logic flows. To those uninitiated into the world of formalized software specification, it might appear as a foreign language using foreign mathematics. But to the software engineering world it is the exact bridge needed to cross from the systems specification to the design process.

10.2.3 Design

Design is generally performed at two primary levels, the overall system design and the design of lower level components, also known as program design. While these two levels are summarized here, it should be remembered that there may be a prototype building cycle in the development plan. In that case, there would be one or more iterations of design-build-test to get a better idea where the system is going, before the final lower design is performed.

10.2.3.1 Systems Level Design

The first design activity is to develop a system level concept of what the software will look like, accomplish, and how it will participate in the overall system of hardware, software, and humans. This includes defining interfaces with the world external to the system, including the user interfaces, the various software subsystems, along with their functions, and the interfaces and data flows between them. The system and software specifications are used extensively to ensure the system is being built to satisfy the requirements. The systems design defines what goes into and out of the system, what functions are performed, and by what subsystems. The design is generally presented for approval or modification at a preliminary or system design review.

10.2.3.2 Program Design

Program design takes the overall structure, functionality, interfaces, and data flows of the system and adds detail. The structure is defined down to the individual program modules. This is done by dividing design into smaller steps, such as *Top Level Design* and *Detailed Design*. Subdividing the design process implements the divide and conquer principle, making design easier and more controlled while facilitating the detection of errors.

All necessary processing and sub-functions that make up the system functions are named, defined, and allocated to specific modules. Because different people may be working on different modules, interfaces must be defined exactly. Data flows and data storage are mapped out and defined in detail. Algorithms, mathematical and other processing are all defined. Individual modules are designed using pseudo-code or natural language to describe what each module does. Data and variables used to interface with other modules are precisely defined.

The completed design is usually presented for formal approval at a critical design review. This is critical as the name says. It is used to spot any inconsistencies, logic errors, interface mismatches, and forgotten functions.

The temptation will always exist to leave more of the design to the coding activity. Remember, "t*he sooner you begin writing code, the longer it will take to get done."* The more complete and detailed the design is, the faster and easier coding can be accomplished. Better design also means fewer errors, easier debugging, and less time spent in the code and test iterations. Good, well-documented designs also make software maintenance much easier and far less costly than it would otherwise be.

10.2.3.3 CASE Tools

Computer Aided Software Engineering (CASE) tools are software programs that assist the software engineer in designing and documenting software. They are usually dependent on the design methodology employed by the development team, and may even be dependent on the language chosen for implementing the design. Their chief contributions consist of the following advantages:

- Encourage or enforce a formal design process or methodology.
- Document the design in a consistent, formal manner.
- Track the details to help ensure things aren't forgotten.
- Unify the development team in their efforts.

In addition to the above advantages, some CASE tools help in discovering errors, developing tests, tracking requirements, and other benefits. Every effort should be made to select an easy-to-use, functional, helpful, and proven tool that not only integrates with your development process, but actually facilitates, assists, and documents that process in as many areas as possible.

10.2.4 Coding with Programming Languages

Coding, or programming, as we have discussed, is only a part of the software engineering process. For most engineers, it is the favorite part, where you put things into the computer and it responds. This is probably the greatest source of temptation to start coding right away, and to try to wrap the other aspects of software engineering into the coding process. Sadly, the desire to move prematurely into programming, to "save time" or "save money," even extends to some leaders and managers, who may be under pressure by those above them who know little of the development process. There is a joke where the manager says to the programmers, "I'll go up and find out what they need and the rest of you start coding."

Programming is where planning, specifying, and designing take on real existence, at least as much as a non-tangible entity can. This is where the engineer creates actual programs that execute on a computer, receiving inputs and providing outputs. Programming starts with coding individual modules or functions. These units are then integrated together in progressively larger and more complex subsystems, until the entire software system is brought together and executed as a system. Each step of integration shows the software units can work together or discovers problems with their interaction. Programming involves these major knowledge areas:

- The programming language
- General programming skills
- The Operating System (OS)
- Hardware that may be associated with the software

10.2.4.1 Programming Languages

There are countless programming languages. They have been developed or created since the beginning of the computer era. Some are general purpose and have been converted to run on many different platforms or computer systems. Others are narrow in their application and have been made for specific purposes or for specific computers. They are called by many names: Cobol, Fortran, C, Basic, APL, Pascal, Ada, C++, Java, FoxPro, C#, Assembly, etc. Each language has specific reserved words it uses to command the computer to perform certain operations. These reserved words must be used in a certain syntax, the order or structure of programming statements, for the computer understand them. In addition, each language has requirements for the structure of the overall program, how it implements general programming operations, how it deals with the computer, and how a program is written, compiled, and executed. Most programmers know more than one language but have a favorite. Learning to use a programming language, like spoken languages, takes training (formal or informal) and practice.

A computer language should be selected for use only after a careful review of the systems and software engineering factors that influence the overall life cycle costs, risks and potential interoperability. [2] When choosing a language, all factors considered should be documented along with the decision process and the results. Major factors to consider are:

- Reliability
- Safety
- Standards
- Development Methodologies
- Cost
- Performance
- Security
- Development Tools
- Maintenance & Supportability
- Schedule
- Interoperability
- System Architecture
- Software Architecture
- Staffing
- Open Systems

10.2.4.2 General programming skills

General programming skills exist apart from specific languages and may be implemented by most languages, albeit in different manners. They include operations (loops, branches, Boolean logic, etc.), structures (arrays, strings, objects, etc.), algorithms (sorting, mathematical, parsing, compression, etc.), data types (strings, integers, floating point, Boolean, etc.), input/output, threads, inheritance, etc. These are learned to some extent by studying theory and to a greater extent by actual programming. Because skills are usually learned to satisfy a need – "How do I sort this efficiently?" – rather than curling up with a good algorithm book in front of the fireplace, experience can play a big part in producing good, efficient program code. This is the area where skilled programmers practice their "art" and employ their tricks.

Another subject belonging to skills is the use of programming style. Just as written language is easier to understand and use when certain style conventions are used, programs can be made easier to follow, understand, debug, test, and maintain if certain style conventions are employed. Generally, this will be one of the project requirements. Various style guides exist for languages in the industry and within many organizations.

10.2.4.3 Operating Systems

The operating system (OS) is a program that acts as the heart and soul of the computer. When a computer is powered up the OS runs through all the system checks and sets up memory and other resources for use by other programs. The OS receives instructions from and presents information to the computer operator and launches other programs as directed. As in programming languages, people prefer one OS over another and often ascribe personality traits to the various OS's they know or have heard of. Unlike languages, which are often associated with divine attributes, OS's are more often than not ascribed diabolical qualities, and viewed as inherently evil entities that must be bypassed or appeased to get anything done.

Software engineers must come to know the OS that resides on the computer for which they are writing software. The OS will in most cases be running at the same time as the new software. In spite of having to figure out ways to work around the OS to accomplish certain software feats, in many cases the OS will perform various functions outside of the new program. For example, why write a driver and interface program for a keyboard, monitor, or printer when one can simply ask the OS to perform that function? Knowing how the OS operates and what it can and cannot do for you is essential for programming.

10.2.4.4 Hardware Systems

Because software cannot be touched, it doesn't exist outside a computer system. Software is meaningless without hardware to exist in. Each type of computer system has its own capabilities, resources, and idiosyncrasies. This is true for all systems, whether they are general-purpose desktop computers or embedded computers in a microwave oven or weapon system. Memory, storage, communications, calculation speed, and interfaces are all constrained and defined by actual physical hardware. To use them, the software engineer must have an understanding of what hardware is available, how it is used or accessed, what its limitations are, and how it is controlled by software. By itself, hardware is just silicon and metal. Software is intangible. Only together do they form a functional system. One cannot be understood without the other.

10.2.5 Testing, Validation, and Verification

Software must be evaluated at intervals during development and when complete to determine if it actually performs as it should. Testing is done at various stages from the coding phase throughout the remainder of the project. Individual programmers test individual modules or units of the program as they are programming. This helps them spot errors in both logic and coding. It also shows that the module is correct and functional. During the integration process, modules are tested together in subsystems to detect errors with module interfacing and with working together. Additionally, integration testing determines whether the larger groupings of modules function smoothly and correctly together. Testing continues with larger and larger groups until the full system is evaluated. If testing is done correctly throughout the development process, the final acceptance testing should be a demonstration of the full, correct functionality of software and its fulfillment of requirements. There should not be any surprise errors or non-performance. The hierarchy of testing is shown in Figure 10-5.

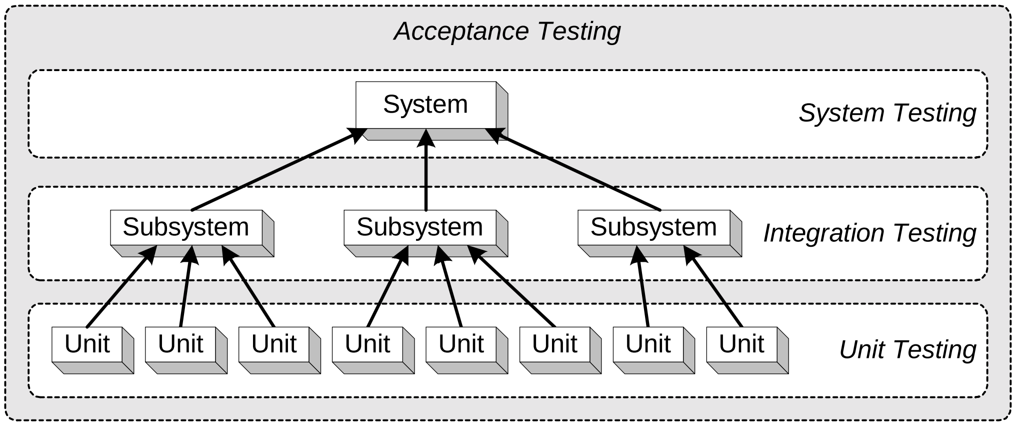

Figure 10-5 Testing Hierarchy

When evaluating software, two specific types of performance are considered: validation and verification. Validation is the determination of whether the right software has been built. Does it meet the requirements established in the beginning phases of the project? Verification is determining whether the software has been built according to the design. A system can be built right without building the right system. In other words, the software may be verified and not validated. Ultimately, the system must meet both criteria, being built according to the design and the requirements, if it is to be useful.

Figures 10-1 and 10-2 show where verification and validation testing are performed and what drives them. It should be noted that although acceptance testing validates the system for the stakeholders, acceptance testing should be a formality. The real testing should have been performed throughout unit testing, integration, and system testing.

10.2.6 Human Factors

While many automated systems deal only with machines, most have at least some interaction with humans. Software engineering includes understanding and taking into account the human aspect of the man-machine interface. This includes human sensory, psychological, and bio-mechanical considerations. Some software functions almost exclusively as an interface to humans. Other software requires human interaction only for startup and occasional direction. Software functionality is worth very little if the user cannot easily and efficiently control the software. Great care and consideration must be given to interaction between the computer and the user. This includes such things as intuitiveness, color, light, noise, redundancy, input methods, and display/output methods of the interface, as well as fatigue, stress, and boredom of the user. Developers must also understand information theory and human information processing. While failure to incorporate human factors into the development process may result in poor sales for a computer game, in a logistics system it can lead to people not using the software tool and bypassing the system. In an avionics or weapons system it can lead to injury or death.

10.2.7 Maintenance

When software is fielded or released to the users there will be a period of training and learning. If the users find things they would like to add or change, and that is likely, the stage is set for upgrades to the software. While we have talked extensively about software development, most of the money spent on software, 70% or more, is spent on maintenance. Maintenance may consist of fixing a problem not discovered in the development phase, adding new functionality to the software, or modifying the software to deal with changes in the rest of the system. While maintenance is usually the longest phase of the software life cycle, it is dealt with in a manner similar to previous development efforts.

Software maintenance projects are development projects that begin with previously constructed software. Using that software becomes one of the requirements of the new, improved system. The maintenance project, like other projects, consists of planning, requirements, design, coding, and testing. Software maintenance will generally go on in a series of discrete upgrade projects as needs arise and as resources are available until the overall system is retired and/or replaced. The maintenance cycle is shown in Figure 10-6.

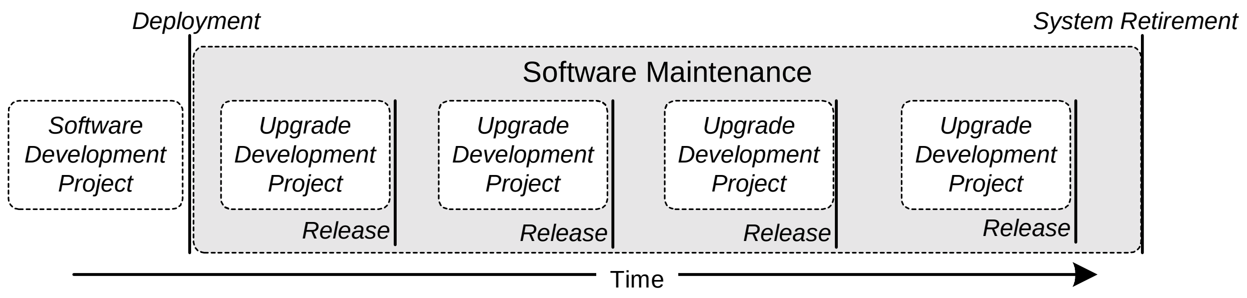
Deployment
System Retirement
Software Maintenance
Software Development Project
Upgrade Development Project
Upgrade Development Project
Upgrade Development Project
Upgrade Development Project
Release
Release
Release
Release
Time

Figure 10-6 Maintenance Upgrade Cycle

10.2.8 Summary

This chapter has provided only a brief overview of primary software engineering activities or knowledge areas. Many other areas, such as those in the list in Section 10.2, have been left out because of time and space constraints, not because they are not essential. Better understanding of the software engineering process can be gained by studying those areas in the reference material listed at the end of this chapter.

Those who become software engineers go through a rigorous course of study, training, and practice to gain knowledge, develop skills, and gain insight into the software development process. The discipline is the application of engineering principles to software development and brings a number of knowledge fields together in a coordinated effort. Because of the vast amount of man years spent developing software there have been many innovations and improvements to the engineering process which have not only benefited the software industry but have migrated into other engineering disciplines as well. Because of the constant improvement of the computer industry and the incorporation of computers into more and more tools, toys, and other systems, keeping up with the advances in software engineering is a full time, and probably lifetime, process.

10.3 Software Engineering Processes Checklist

This checklist is provided to assist you in understanding the software engineering issues of your project. If you cannot answer a question affirmatively, you should carefully examine the situation and take appropriate action.

10.3.1 Before Starting

- ❑ 1. Do you know what software development life cycle your project will be employing and how it coordinates with the software and project life cycles?
- ❑ 2. Does the development team have experience in the software development life cycle to be used?
- ❑ 3. Do the developers, the stakeholders, and you understand what the steps of the development process are, and what the inputs and products of each step are?
- ❑ 4. Has your project been planned in the various software development areas listed in Section 10.2.1 and in Chapter 3?
- ❑ 5. Do you know what design method has been chosen for the development effort and why it was chosen over other methods?
- ❑ 6. Does your development team have experience with the chosen design method? If not, has the schedule been adjusted to allow for learning the new design method?
- ❑ 7. Have proven CASE tools been chosen to assist in the software design?
- ❑ 8. Does your development team have experience with the chosen CASE tools? If not, has the schedule been adjusted to allow for learning to use the CASE tools?
- ❑ 9. Has an appropriate programming language been chosen and do you know the reasons it was chosen?

❑ 10. Does your development team have experience with the chosen programming language? If not, has the schedule been adjusted to allow for learning the chosen programming language?

❑ 11. Is the development team sufficiently skilled and experienced in programming to properly and efficiently design, code, and test the software?

10.3.2 During Project Execution

❑ 12. Are your requirements complete, unambiguous, and agreed to by both developers and stakeholders?

❑ 13. Have you completed both system and functional specifications, and have they been reviewed and approved by stakeholders?

❑ 14. Is the development team familiar with or provided with the appropriate opportunity to become familiar with the operating system and system hardware?

❑ 15. Is a detailed software design being completed, reviewed, and approved before coding starts?

❑ 16. Is testing being properly implemented and satisfactorily completed at unit, integration, and system levels before acceptance testing?

❑ 17. Are human factors being considered sufficiently in the software design?

10.3.3 At Completion

❑ 18. Does the completed software correctly implement the design?

❑ 19. Does the software meet the requirements?

❑ 20. Does the software meet the users' needs?

10.4 References

[1] Michael Black, "Introduction to Software Engineering" Lectures: www.cs.brown.edu/courses/cs032/

[2] DoD 5000.2-R, Mandatory Procedures For Major Defense Acquisition Programs (MDAPS) And Major Automated Information System (MAIS) Acquisition Programs, April 5, 2002. Section 4.3.5: http://sw-eng.falls-church.va.us/dod5000-2.html

10.5 Resources

ASD(C3I) Memorandum, "Use of the Ada Programming Language," April 29, 1997. Factors to consider when choosing selecting a programming language: http://sw-eng.falls-church.va.us/oasd497.html

Crosstalk Magazine: www.stsc.hill.af.mil/crosstalk/

– "Getting Software Engineering into Our Guts": www.stsc.hill.af.mil/crosstalk/2001/jul/bernstein.asp
– "The Software Engineer: Skills for Change": www.stsc.hill.af.mil/crosstalk/2001/jun/cross.asp
– "The V Model: www.stsc.hill.af.mil/CrossTalk/2000/jun/hirschberg.asp

DeGrace, Peter and Stahl, Leslie, *Wicked Problems, Righteous Solutions: A Catalogue of Modern Software engineering Paradigms*, Yourdon Press.

Department of Energy (DOE) *Software Engineering Methodology*: http://cio.doe.gov/sqse/sem_toc.htm

Guide to Software Engineering Body of Knowledge: www.swebok.org

Guidelines for the Successful Acquisition and Management of Software-Intensive Systems (GSAM), Version 3.0, Chapter 11, OO-ALC/TISE, May 2000. Available for download at: www.stsc.hill.af.mil/gsam/guid.asp

Program Manager's Guide for Managing Software, 0.6, 29 June 2001: www.geia.org/sstc/G47/SWMgmtGuide%20Rev%200.4.doc

Software Engineering Body of Knowledge:
www.sei.cmu.edu/publications/documents/99.reports/99tr004/99tr004abstract.html

Software Engineering Education websites: http://faculty.db.erau.edu/hilburn/se-educ/

Software Engineering Institute: www.sei.cmu.edu

Software Engineering Process Group, Tutorials: http://prg.cpe.ku.ac.th/developer/tutorial.html

Software Reality. Stories of software engineering mistakes: www.softwarereality.com

University of Michigan, Introduction of Software Engineering:
www.engin.umd.umich.edu/CIS/course.des/cis375.html

University of Toronto, Software engineering notes: www.cs.toronto.edu/~sme/CSC444F/

Verification, Validation and Evaluation of Expert Systems: www.tfhrc.gov/advanc/vve/cover.htm

This page intentionally left blank.