PDHonline Course E145 (2 PDH)

# Software Engineering Concepts

*Instructor: Warren T. Jones, Ph.D., PE*

2020

PDH Online | PDH Center

5272 Meadow Estates Drive
Fairfax, VA 22030-6658
Phone: 703-988-0088
www.PDHonline.com

An Approved Continuing Education Provider

# Software Engineering Concepts

*Warren T. Jones, Ph.D., P.E.*

## Course Content

## Table of Contents

## Module #1: Introduction and Definitions

IEEE software engineering definition:
Software engineering is the application of a systematic disciplined quantifiable approach to the development, operation and maintenance of software; the application of engineering to software.

Issues that have led to the development of current software engineering practice are listed below. These problems remain as challenges today, especially for large-scale software development.

- Difficulty of software projects meeting delivery deadlines
- High cost of software development
- Errors in delivered software after extensive testing
- High cost in time and effort in maintaining software
- Problems measuring progress during software development and maintenance

The above issues are the result of the fact that building software is inherently different from hardware or any other engineering system. It is important to understand the characteristics of software that underlie these differences. Some important ones are as follows:

- Software is not manufactured in the traditional sense since it is intangible.
- Hardware systems wear out. Software does not.
- The software industry lacks the well-developed design handbooks and components-based development of other engineering disciplines.

In order to address the software development problems cited above and to reduce the potential chaos of developing an intangible product, a framework called a software process is needed for the tasks that are required for the building of high-quality software. The process model used on a given project will depend on the nature of the project. However, the following basic activities are common to all process models:

- Software specification – functional requirements obtained from the user
- Software design and implementation – production of the software system as a product
- Software validation – activity that assures that customer specifications are met
- Software evolution – system modification to meet continuing customer needs

## Module #2 Software Process Models

A number of process models have been developed. Each emphasizes different aspects of the software life cycle and each will be appropriate for projects for which the emphasized aspects are important.

## The Waterfall Model

As the original software process model, it can be viewed as a "first approximation" of the activities needed in the software development process. Some versions provide feedback loops from each stage to the previous ones, but it is most used as a simple linear model as shown in Figure 1. In reality it is only suitable for projects in which all the customer requirements are known at the outset of the project, a rare condition, even for small scale projects. Also note that a working version of the system is only available late in the project, a problem addressed by the incremental process models.

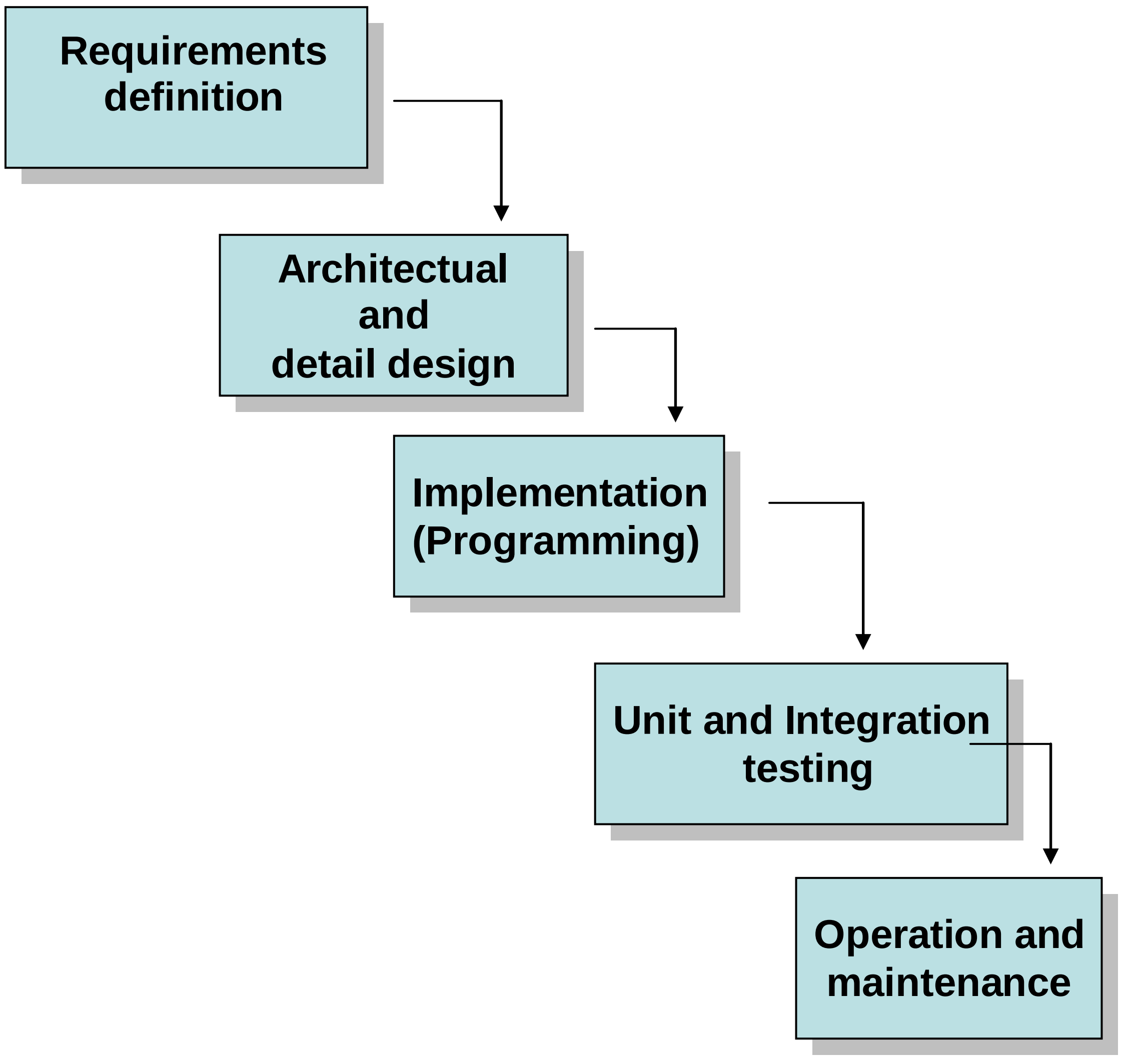
Requirements definition
Architectual and detail design
Implementation (Programming)
Unit and Integration testing
Operation and maintenance

**Figure 1 – Waterfall Model**

## Incremental Process Models

These models add flexibility to the basic waterfall model by adding incremental iteration processes. For example, each function of the system can be treated as a separate system and developed in accordance with the waterfall model. With this approach the system can be delivered to the customer incrementally beginning with the first function and later functions one by one on a different project schedule, with each function delivered initially as a separately operating function.

A variation of this incremental theme, called the Rapid Application Development (RAD), applies the software specification stage to the entire system but for software design and implementation stages the system is divided into team projects to enable development time to be scaled down to the range of two to three months. RAD may not be an appropriate choice if the customers and developers are not prepared for the rapid pace or if the system does not lend itself to the modularization required for assignment to teams.

## Evolutionary Process Models

These models are designed to grow the final software system by iterative cumulative development. For example, **Rapid Prototyping** is sometimes used at the beginning of projects to obtain improved understanding of the customer requirements. The prototype can either be a throw-away or can be extended to the development of the entire system. **The Spiral Model** combines elements of the waterfall model and rapid prototyping to implement evolutionary development as shown in Figure 2 below. Each traversal around the spiral, beginning with Objectives, represents a new more complete version of the system with a risk assessment each time around. Each version can be viewed as a system prototype during any phase of the evolutionary development. For example, one of the spiral traversals might represent the system design and another might focus on integration testing.

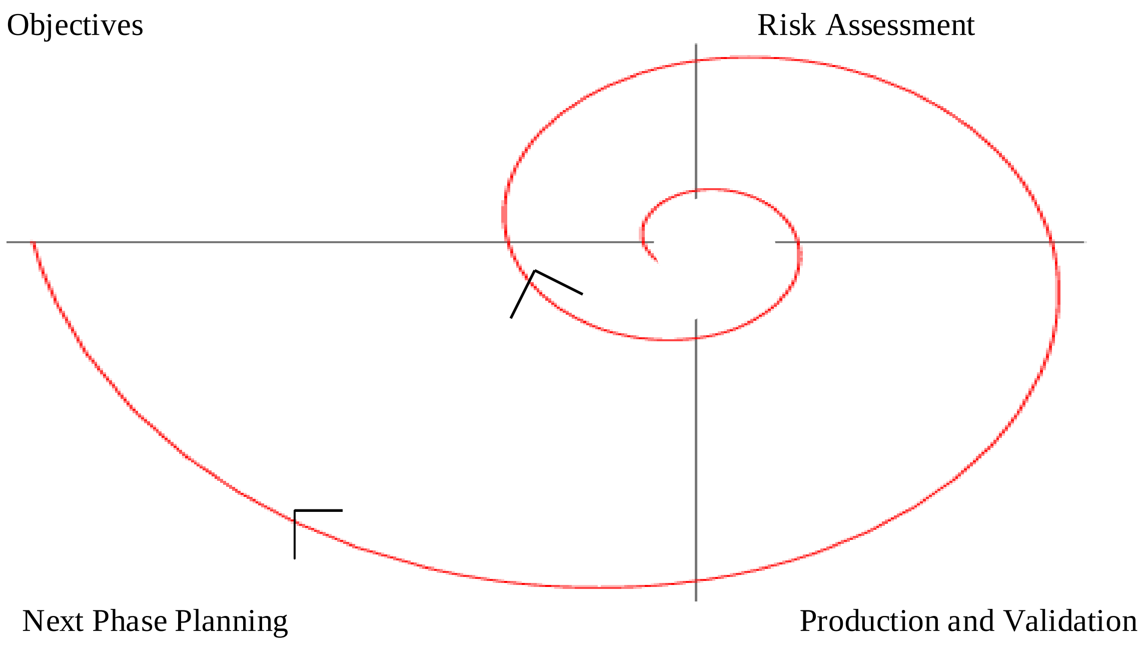

**Figure 2 – The Spiral Model**

In the **Concurrent Development Model,** all activities in the process model exist concurrently in various states such as "awaiting changes", "none" and "under development". This model describes the software process to be implemented as a network of activities rather than a simple linear process. **Component-Based Development** uses off-the-shelf software packages as the defining characteristic of the process. This is a software reuse approach and often leads to large reductions in cost and development time but depends on the nature and quality of the components library available.

### Aspect-Oriented Software Development (AOSD)

ASOD is a new concept intended to model localized features, functions and information content that have impact across the software system. It is sometimes called crosscutting concerns that go beyond mechanisms such as subroutines.

### Agile Development

A somewhat controversial approach to the software process is to compress and overlap the traditional life cycle phases as much as possible with close customer partnership. For more details on philosophy, see the Manifesto for Agile Software Development. The overriding objective is rapid time to delivery. “Extreme Programming” is also a term that is associated with this strategy. Some have said that agile development models are appropriate for web applications developers, a group that has tended to resist the discipline of software processes.

## Module #3 Software Process Activities

### Requirements Engineering

Requirements engineering is about communicating with the customer. The objective is to arrive at a written agreement describing the functionality of the software to be developed. The final product of this activity is usually called a specification and forms the basis for all the development activities that follow. It turns out that the activities of this stage of the development life cycle are among the most difficult, but perhaps the most important.

The problem of developing a clear understanding of what the customer needs has become a classic challenge. Newcomers to the field are sometimes surprised to learn that the customer representatives themselves find it difficult to communicate their needs clearly. Given this state of affairs, projects frequently find themselves changing requirements throughout the development period. The later in the development process that a requirement is changed, the more difficult and more expensive that change becomes. Therefore, what may seem to be excessive time spent on this first phase is in fact an excellent investment in risk reduction for the entire project. Many tools are used at this stage to increase confidence in understanding what the system should be like, including rapid prototyping, user scenarios and functions/feature lists. Bringing modeling and design tools into this phase is not unusual. Not surprisingly, this phase is characterized by lengthy meetings between the developer and customer.

## Requirements Analysis and Modeling

The focus of this activity is on defining the operational characteristics of the software and has three primary goals of providing the following:

- Behavioral description of customer requirements with the emphasis on the *what* rather than the *how*
- Foundation for the software design
- Operational system definition that can be used for system validation after the software development is complete.

The following subactivities are identified with requirements analysis and modeling:

### Domain analysis

Reusability is an important goal in software development since it reduces development costs, increases reliability and reduces development time. Domain analysis is the process of identifying patterns that can be reused. These patterns can be any common functions or features that have the potential for broad use across an application domain. An application domain is typically a class of problem such as financial, medical or aerospace; however, the broader the reuse the better.

### Data modeling

Analysis modeling sometimes begins with the identification of all data objects that are to be processed in the system and the relationships between these objects. Data modeling is used for large database and information systems applications.

### Object oriented analysis

The object-oriented (OO) approach to analysis represents the latest "paradigm shift" in analysis methodology and is epitomized by the Java language at the implementation stage. Some of the claims which have made this approach popular are as follows:

- Customers can understand OO models with no programming knowledge thus facilitating the all-important early phases of communication.
- OO languages promote code reuse and thus programmer productivity
- The OO design and analysis methods are accommodating to change

The OO approach is based on modeling of the problem domain using classes and objects.

<u>Class</u>: defines the data and procedural abstractions for the information content and behavior of some system entity.

<u>Method</u>: representation of one of the behaviors of a class.

Object: instance of a specific class. Objects can inherit the attributes and operations defined for a class. Classes are sometimes illustrated as “cookie cutters” and the associated objects as “cookies”.

The goal of object-oriented analysis is the design of all classes and associated methods that are appropriate for the system being developed.

The unified modeling language (UML) has been developed for the modeling and development of object-oriented (OO) systems. UML has become an industry standard for OO development.

### Scenario-based modeling

End-user involvement in a software project is critical to its success. Scenario-based modeling provides mechanisms for capturing information on how end-users desire to interact with the system. UML provides support for the development of interaction scenarios that begin with the writing of use-cases that describe a use of the system by a specific end-user. The dynamics of these use-cases can be represented in UML activity diagrams similar to flow charts. More complex interactions can be captured in UML swimlane diagrams that can model concurrent activities.

### Flow oriented modeling

Although not part of UML, the input-process-output data flow diagrams (DFD) continue to be a very popular analysis modeling tool and can be used to augment UML diagrams. Data flow in the system can be modeled in a hierarchical fashion with DFDs with higher level context diagrams being refined with greater detailed DFDs at lower levels.

### Dynamic modeling

After static data and attribute relationships have been established, it is useful to create behavioral models to represent the systems response to external events. Use-cases can be used to identify events and UML sequence diagrams can be used to model how events trigger transitions from one object to another.

## Architectural Engineering and Design

The design activity is the bridge between the software requirements and analysis models, and deliverable product construction. Design is the process of producing the “blueprint” for the coding and testing. It is also the activity which establishes software quality. The results of design activities are representations which can be assessed for quality. The list of frequently cited software quality attributes is sometimes called FRUPS, an acronym for the following list:

- Functionality
- Reliability
- Usability
- Performance
- Supportability

There is a huge difference between simply getting code to work and engineering a system of high quality. The following design concepts have been helpful in achieving software quality:

**Abstraction**

When developing an architectural design of a complex system, many levels of abstraction are needed to describe the system. Higher levels contain fewer details and lower levels provide increasingly more system information. Procedural abstractions contain instructions but suppress details. Data abstractions refer to data objects and their properties.

**Modularity**

When dealing with complexity, cognitive psychologists tell us that we humans can deal with only from five to nine chunks of information at a time. Therefore, the strategy of designing a system as a collection of integrated modules is necessary for system understanding.

**Information Hiding**

Many architectural designs are possible for a given project. How does one assess the quality of a given system modularization? Application of the principle of information hiding in the development a modularization is known to increase quality as defined by the FRUPS attributes. The idea is to define module boundaries so that local information is encapsulated and hidden from its outside world. Module interfaces are designed to communicate only information that is essential to invoking the functionality of the module. Careful application of this principle pays large dividends during the testing and maintenance phases of the life of the system.

**Functional Independence**

Modules that are designed to be functionally independent and have simple interfaces with the remaining system are known to be easier to develop, test and maintain. Two criteria for assessing independence are cohesion and coupling, measures of singleness of function and intermodule connectivity, respectively. High cohesion and low coupling contribute to higher quality.

## Refinement

The design process is sometimes called a top-down step-wise refinement of the top level system abstraction to successive lower levels of abstraction by the application of the above modularization principles. Modules are created by hierarchical decomposition.

## Refactoring

This activity is usually specific to agile methods. It refers to the internal redesign or restructuring of a component or subsystem in ways that improve its quality and performance.

## Design and Reuse of Patterns

As with other more mature engineering disciplines, one should always approach design decisions with the mindset that design patterns used in the past should be considered first rather than proceeding with a design derived from the uniqueness of the requirements of the particular project. If patterns of the past do not seem to be suitable, creating new ones should be the next level of consideration and contributing to the pattern library for the use of future projects. Design patterns range from the architectural level down to component detail design.

## Component Level Design

This level of design describes the data structures, interfaces and algorithms. Component level design can be represented in a programming language, but is also often described in some other intermediate representation such as a program design language (PDL) for conventional module design and the Object Constraint Language (OCL) in the object-oriented design world.

## User Interface Design

A common failure of software projects is to spend too little time communicating with the user. It is easy for software experts to fall into the subconscious trap of “knowing what is good for the user”. What may seem to be “clearly good for the user” is all too frequently not the case from the perspective of the user herself. The use of user scenarios and very early and iterative prototype screen designs can help to assure that the user is being understood. It has been said that you should plan on building one to throw away. Three good guidelines are the following:

- Put the user in control
- Reduce the user’s memory load
- Make the interface consistent

## Software Testing

After the software system is coded into a deliverable product, testing strategies are used to validate system requirements. Testing strategies are designed to detect errors in the system. Debugging is the process of finding the source of the errors for correction. Exhaustive testing is impractical. Therefore, no matter how much testing is done, it is never known with certainty if all bugs have been detected. Since testing is a process of detecting the presence of errors, the absence of all errors cannot be guaranteed by the testing process. A high percentage of project resources are expended on the testing phase. Testing usually proceeds in two phases, first at the component level sometimes called unit testing. Unit testing is followed by integration testing in which increasingly larger groups of components are tested culminating in the total system. Unit testing is usually done by the developer and integration testing by an independent test group. Testing strategies for conventionally designed software differ somewhat from those for object-oriented systems.

**Conventional software**: unit testing focuses on execution paths through component program logic with the goal of maximizing error detection by path coverage; whereas integration testing usually involves input and output values.

**Object-oriented software**: unit testing is done with classes, whose definition involves not only internal program logic but also attributes and operations as well as communication and collaboration. Operations must be tested in the context of a class. Two approaches to integration testing of object-oriented systems are common, thread-based and use-based testing. The thread-based approach tests the set of classes that respond to a given system input or event. Use-based testing begins with by testing classes that are relatively independent of all others and continues in stages with each stage defined by the addition of a layer of dependent classes until the entire system is encompassed.

After unit and integration testing, the entire system is tested in accordance with customer requirements. This final testing phase is usually called validation testing and includes alpha and beta tests. Alpha tests are performed at the developer site and beta tests occur later at end user sites. Final release of the software is scheduled after the beta tests are complete.

## Product Metrics for Software

The use of objective measures of software development products as an empirical measure of quality is somewhat controversial in the software engineering community. Some say that our lack of basic understanding of software justifies delaying the development and use of such metrics. However, many metrics are available to help assess and guide analysis, design, source code development and testing. Some examples are given below:

Analysis: overall system size metric defined as a function of information in the analysis model.

Design: component level metrics that measure complexity

Source code: length metric defined in term of lines of code (LOC).

Testing: coverage of the program as a directed graph

# Module #4 Software Project Management

In many ways managing a software project is like managing any other engineering project. However, it is also true that software project management is more difficult. Perhaps the most important reason is the product is intangible. Monitoring the completion status of an entity that one cannot see or feel is a formidable challenge. Also standard processes and designs do not exist in the software field in the same way that they are found in handbooks of other engineering disciplines. People are the crucial element in project management and can be organized in teams that vary in their level of autonomy from traditional hierarchical structures to the "self-organizing" teams of the new agile paradigm.

## Project Estimation

Early in a project the software development group and management must establish estimates for resources required, work to be done and time to delivery. Project planning is crucial to success. Technical people frequently do not take planning activities as seriously as they should and project cost, quality, and time to delivery are often affected as a result. Cost estimating techniques are available based on metrics accumulated from past similar project experiences. The usual approach is to use several methods and compare values. If these values vary widely, then this variance is taken as an indication of the need for more information. COnstructive COst MOdel (COCOMO) II is a popular estimation model for conventional software. The parameters in this model have been derived from data from over 4,000 software projects. Estimation methods have also been developed for object-oriented and agile development.

An important management observation is that software development time is not solely a function of the number of people on the project. One should not succumb quickly to the temptation to add more people to a late project. Adding more people could actually make it later.

## Risk Management

Anticipating and having a plan for potential project problems will help avoid "crisis management" when problems do occur. The Spiral Process Model discussed earlier provides a framework for dealing with this issue. An investment of project management time in risk identification and monitoring can help keep potential problems to a minimum. Some examples of project risks are: changing requirements, low estimates of components reuse, high technical staff turnover and change in delivery deadline.

### Capability Maturity Model Integration ( CMMI )

CMMI was originally developed for the Department of Defense to assess the quality and capability of software contractors. Even though the models developed have been extended to help any organization improve and measure their capabilities, these resources are worth exploring and understanding.

### IEEE Certified Software Development Professional Program ( CSDP )

IEEE offers three software development certifications that can be valuable in the marketplace. These certification opportunities could be a natural next step after this concepts-oriented software engineering course.

- Associate Software Developer – basic knowledge required for software product development.
- Professional Software Developer – focuses on competency in four important areas:
    - Software Requirements
    - Software Design
    - Software Construction
    - Software Testing
- Professional Software Engineering Master – includes mastery of 11 key areas of software engineering.
    - Software Requirements
    - Software Design
    - Software Construction
    - Software Testing
    - Software Maintenance
    - Software Configuration Management
    - Software Engineering Management
    - Software Engineering Process
    - Software Engineering Models and Methods
    - Software Quality
    - Software Engineering Economics

# Module #5 Software Security

Everyone is aware that security is one of the most important issues in the computer field today. What is not apparent to everyone is that the security challenges today are frequently software problems. The weak points are the applications at the ends of the communications link and therefore represent the points of greatest vulnerability to attack.

Three trends are often cited as introducing security risks into systems and contributing to the magnitude of the security problem today.

1. The increasing complexity of systems make them more difficult to understand and hence more difficult to secure.
2. Increasing access to applications through various computer network technologies adds considerably to the security risks.
3. Software is being increasingly designed to be extensible with the incremental addition of functionality making it impossible to anticipate the kind of mobile code (updates) that may be downloaded.

## Can Security be Defined?

A good question is "Can we ever declare a software application secure?". Unfortunately, security, like many other engineering goals, is a relative quantity and 100 percent security is unachievable. A better question is to be more specific and ask "Secure against what and from whom?". Some consider security to be a subset of reliability.

## Approaches to the Security Problem

### Penetrate and Patch

Often software is developed in an "Internet time" highly compressed schedule in order to be first to market. This approach considers security as an add-on feature after delivery. When vulnerabilities are found, frequently as a result of an attack, patches are developed and issued to the user community. There are many problems with this "penetrate-and-patch" approach to security. Here are a few:

- "Developers can only patch problems which they know about. Attackers may find problems that they never report to developers.
- Patches are rushed out as a result of market pressures on vendors, and often introduce new problems of their own to a system.
- Patches often only fix the symptom of a problem, and do nothing to address the underlying cause.
- Patches often go unapplied, because system administrators tend to be overworked and often do not wish to make changes to a system that "works [9]". It should also be noted that system administrators are often not security experts.

### Build Security into the Software Development Life Cycle

The recommended approach is to incorporate software security as an engineering goal throughout the software engineering life cycle. Since many of the issues of software security are issues of risk management, the spiral model of software development is often mentioned as appropriate, with the repetitive spiral refining and converging security considerations toward the final goal. Some activities that should be added to each life cycle stage are listed below.

Requirements: Add security specifications.

Design: Develop threat models by viewing the system form an adversary's perspective [8] and apply security design principles, e.g. "Design with the Enemy in Mind" [9].

Implementation: Add secure coding standards and language subsets

Testing: Add Security test plans and use random input testing (e.g. fuzz testing or vulnerability analysis using penetration testing.

## Principles for Software Security

It has been said that 90% of security problems can be avoided if the following principles are followed [12]:

1. Secure the weakest link: security is a chain

2. Practice defense in depth: manage risk with diverse defensive strategies

3. Fail securely: Failures are unavoidable and should be planned for

4. Follow the principle of least privilege: minimum access required to perform an operation and only for the minimum time necessary

5. Compartmentalize: minimize the amount of potential damage by organizing the system into the smallest number of units as possible.

6. Keep it simple

7. Promote privacy

8. Remember that hiding secrets is inherently difficult

9. Be reluctant to trust: Servers should be designed to distrust clients and conversely.

10. Use your community resources: Use security libraries and cryptographic algorithms that have been widely used and evaluated

## Language Selection

Many factors influence the choice of a programming language to use for implementation. It is common for efficiency considerations to dominate the language selection process. One of the factors should be security considerations. For example, choosing the C programming language for efficiency should take into account the inherent security risks associated with a language that has no bounds checks on array and pointer references. The programmer must build these checks into the program code. C program efficiencies and low-level data manipulation capabilities come at the high risk of security vulnerabilities and very special diligence is required. Using a language like Java can greatly reduce these risks, since it performs bounds checking. However, the system requirements must tolerate a lower level of run-time performance for this to be a viable option.

Buffer overflows as a security vulnerability have been discussed for many years and yet this type of software problem continues to be a frequently reported instance of system attacks. A buffer overflow is a condition caused by a write operation into a fixed-sized buffer in which the size of the data is larger than the size of the buffer. Most buffer overflows are the result of the properties of the C language mentioned above. This is the case with C++ as well.

**Software Engineering Institute** (SEI )

The mission of SEI is to research complex software engineering, cybersecurity and artificial intelligence problems; create and test innovative technologies and transition maturing solutions into practice. The **Department of Defense,** as well as **other public agencies** and **private businesses**, can meet mission goals and gain strategic advantage by using tools, technologies, and practices developed or matured by the SEI. Maintaining contact with this resource is a good investment.

**National Institute of Standards and Technology (NIST) National Vulnerability Database ( NVD )**

The NVD is a U. S. government repository which includes vulnerability management data including security-related software flaws, misconfigurations and impact metrics. Common Vulnerabilities Exposures (CVE) defines a vulnerability as "a weakness in the computational logic (code) found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity or availability. Mitigation of the vulnerabilities in this context typically involves coding changes, but could also include specification changes or even specification deprecations (e.g. removal of affected protocols or functionality in their entirety)."

"The Common Weakness Enumeration Specification (CWE) provides a common language of discourse for discussing, finding and dealing with the causes of software vulnerabilities as they are found in code, design, or system architecture. Each individual CWE represents a single vulnerability type."

Examples of vulnerabilities can be explored by clicking CWE and scrolling up to a table displaying a list. By clicking on each CWE-ID, details about that vulnerability can be explored. Information includes: Extended Description, Common Consequences, Impact, Observed Examples and References in the technical literature for more details about the vulnerability.

The NIST NVD is a valuable resource for learning about vulnerabilities. You may want to make use of the FAQ page to help you understand and navigate around the website.

## Additional Resources

1. Graff, Mark G. and Kenneth R. van Wyk, **Secure Coding: Principles and Practices**, O'Reilly and Associates, Inc., 2003.

2. Harrington, Ted, **How to Do Application Security Right**, Lioncrest Publishing, 2020.

3. Pressman, Roger S. and Bruce Maxim, **Software Engineering: A Practitioner's Approach,** 8th Edition, McGraw-Hill Education, 2014.

4. Schneier, Bruce, **Secrets and Lies: Digital Security in a Net,** Wiley, 2015.

5. Seacord, Robert C., **Secure Coding in C and C++ (SEI Series in Software Engineering),** Addison-Wesley Professional, 2013.

6. Shostack, Adam, **Threat Modeling: Designing for Security**, Wiley, 2014**.**

7. Sommerville, I., **Engineering Software Products: An Introduction to Modern Software Engineering**, Pearson, 2019.

8. Swiderski, Frank and Window Snider, **Threat Modeling**, Microsoft Press, 2004.

9. Graff, Mark G. and Kenneth R. van Wyk, **Secure Coding: Principles and Practices**, O'Reilly and Associates, Inc., 2003.

10. Thayer. R. and M. Christensen, (Editors), **Software Engineering Volume 2 – The Supporting Process,** 2nd Edition, Wiley-IEEE Computer Society Press, 2002.

11. Thayer, R. and M. Dorfman (Editors), **Software Engineering Volume 1 – The Development Process,** 2nd Edition, Wiley-IEEE Computer Society Press, 2005.

12. Viega, John and Gary McGraw, **Building Secure Software: How to Avoid Security Problems the Right Way**, Addison-Wesley, 2002.