CHAPTER

9 Domain 8: Application development security

EXAM OBJECTIVES IN THIS CHAPTER

- Programming Concepts
- Application Development Methods
- Object-Oriented Design and Programming
- Software Vulnerabilities, Testing, and Assurance
- Databases
- Artificial Intelligence

UNIQUE TERMS AND DEFINITIONS

- Extreme Programming (XP)—an Agile development method that uses pairs of programmers who work off a detailed specification
- Object—A "black box" that combines code and data, and sends and receives messages
- Object-Oriented Programming—changes the older procedural programming methodology, and treats a program as a series of connected objects that communicate via messages
- Procedural languages—programming languages that use subroutines, procedures and functions
- Spiral Model—a software development model designed to control risk
- Systems Development Life Cycle—a development model that focuses on security in every phase
- Waterfall Model—An application development model that uses rigid phases; when one phase ends, the next begins

INTRODUCTION

Software is everywhere: not only in our computers, but in our houses, our cars, and our medical devices, and all software programmers make mistakes. As software has grown in complexity, the number of mistakes has grown along with it. We will learn in this chapter that programmers may make 15-50 mistakes per thousand lines of code, but following a programming maturity framework such as the SEI

CISSP® Study Guide. DOI: 10.1016/B978-1-59749-563-9.00009-3
© 2010 Elsevier, Inc. All rights reserved.

Capability Maturity Model (CMM) can lower that number to 1 mistake per thousand. That sounds encouraging, but remember that an operating system like Microsoft Vista has 50 million (50,000,000) lines of code.

As our software has grown in complexity, the potential impact of a software crash has also grown. Many cars now use "fly by wire" (software) to control the vehicle: in that case, the gear shift is no longer directly mechanically connected to the transmission; instead, it serves as an electronic input device, like a keyboard. What if a software crash interrupts I/O?

Developing software that is robust and secure is critical: this chapter will show how to do that. We will cover programming fundamentals such as compiled versus interpreted languages, as well as procedural and object-oriented programming languages. We will discuss application development models such as the *Waterfall Model*, *Spiral Model*, and *Extreme Programming* (XP) and others. We will describe common software vulnerabilities, ways to test for them, and maturity frameworks to assess the maturity of the programming process and provide ways to improve it.

PROGRAMMING CONCEPTS

Let us begin by understanding some cornerstone programming concepts. As computers have become more powerful and ubiquitous, the process and methods used to create computer software has grown and changed. Keep in mind that one method is not necessarily better than another: As we will see in the next section, high-level languages such as C allow a programmer to write code more quickly than a low-level language such as assembly, but code written in assembly can be far more efficient. Which is better depends on the need of the project.

Machine Code, Source Code, and Assemblers

Machine code (also called machine language) is a software that is executed directly by the CPU. Machine code is CPU-dependent; it is a series of 1s and 0s that translate to instructions that are understood by the CPU. *Source code* is computer programming language instructions which are written in text that must be translated into machine code before execution by the CPU. High-level languages contain English-like instructions such as "printf" (print formatted).

Assembly language is a low-level computer programming language. Assembly language instructions are short mnemonics, such as "ADD," "SUB" (subtract), and "JMP" (jump), that match to machine language instructions. An assembler converts assembly language into machine language. A *disassembler* attempts to convert machine language into assembly.

Compilers, Interpreters, and Bytecode

Compilers take source code, such as C or Basic, and compile it into machine code.

Here is an example C program "Hello World":

```
int main()
{
printf("hello, world");
}
```

A compiler, such as gcc (the GNU Compiler Collection, see http://gcc.gnu.org) translates this high-level language into machine code, and saves the results as an executable (such as "hello-world.exe"). Once compiled, the machine language is executed directly by the CPU. hello-world.exe is compiled once, and may then be run countless times.

Interpreted languages differ from compiled languages: interpreted code (such as shell code) is compiled on the fly each time the program is run. Here is an example of "Hello World" program written in the interpreted scripting language Perl (see: http://www.perl.org):

```
#!/usr/local/bin/perl
print "Hello World\n";
```

This code is saved as "hello-world.pl." Each time it is run, the Perl interpreter (located at/usr/local/bin/perl in the previous code) translates the Perl instructions into machine language. If hello-world.pl is run 100 times, it will be compiled 100 times (while hello-world.exe was only compiled once).

Bytecode, such as Java bytecode, is also interpreted code. Bytecode exists as an intermediary form (converted from source code), but still must be converted into machine code before it may run on the CPU. Java Bytecode is platform-independent code which is converted into machine code by the Java Virtual Machine (JVM, see Chapter 6, Domain 5: Security Architecture and Design for more information on java bytecode).

Procedural and Object-Oriented Languages

Procedural languages (also called *procedure-oriented languages*) use subroutines, procedures, and functions. Examples include Basic, C, Fortran, and Pascal. Object-oriented languages attempt to model the real world through the use of objects which combine methods and data. Examples include C++, Ruby, and Python; see the "Object Orientation" section below for more information. A procedural language function is the equivalent of an object-oriented method.

The following code shows the beginning "ram()" function, written in C (a procedural language), from the BSD text-based game *Trek*.

```
void
ram(ix, iy)
int ix, iy;
{
    int i;
    char c;
```

```
        printf("\07RED ALERT\07: collision imminent\n");
        c = Sect[ix][iy];
        switch (c)
        {
        case KLINGON:
            printf("%s rams Klingon at %d,%d\n", Ship.shipname, ix, iy);
            killk(ix, iy);
            break;
        case STAR:
        case INHABIT:
            printf("Yeoman Rand: Captain, isn't it getting hot in here?\n");
            sleep(2);
            printf("Spock: Hull temperature approaching 550 Degrees Kelvin.
\n");
            lose(L_STAR);
        case BASE:
            printf("You ran into the starbase at %d,%d\n", ix, iy);
            killb(Ship.quadx, Ship.quady);
            /* don't penalize the captain if it wasn't his fault */
```
[1]

This ram() function also calls other functions, including killk(), killb(), and lose().

Next is an example of object-oriented Ruby (see: http://ruby-lang.org) code for a text adventure game that creates a class called "Verb," and then creates multiple Verb objects. As we will learn in the "Object Orientation" section below, an object inherits features from its parent class.

```
    class Verb
        attr_accessor:name,:description
        def initialize(params)
            @name = params[:name]
            @description = params[:description]
        end
    end
    # Create verbs
    north = Verb.new(:name => "Move east",:description => "Player moves to
the north")
    east = Verb.new(:name => "Move east",:description => "Player moves to
the east")
    west = Verb.new(:name => "Move east",:description => "Player moves to
the west")
    south = Verb.new(:name => "Move east",:description => "Player moves to
the south")
    xyzzy = Verb.new(:name => "Magic word",:description => "Player
teleports to another location in the cave")
```
[2]

Note that coding itself is not testable; these examples are given for illustrative purposes.

Fourth-generation Programming Language

Fourth-generation programming languages (4GL) are computer languages that are designed to increase programmer's efficiency by automating the creation of computer programming code. They are named "fourth generation" because they can be viewed as the fourth step of evolution of computer languages:

- First-generation language: machine code
- Second-generation language: assembly
- Third-generation language: COBOL, C, Basic
- Fourth-generation language: ColdFusion, Progress 4GL, Oracle Reports

Fourth-generation languages tend to be Graphical User Interface (GUI)-focused; dragging and dropping elements, and then generating code based on the results. 4GL languages tend to be focused on the creation of databases, reports, and websites.

Computer-Aided Software Engineering (CASE)

Computer-Aided Software Engineering (CASE) uses programs to assist in the creation and maintenance of other computer programs. Programming has historically been performed by (human) programmers or teams: CASE adds software to the programming "team."

There are three types of CASE software:

1. "Tools: support only specific task in the software-production process.
2. Workbenches: support one or a few software process activities by integrating several tools in a single application.
3. Environments: support all or at least part of the software production process with a collection of Tools and Workbenches."[3]

Fourth-generation computer languages, object-oriented languages, and GUIs are often used as components of CASE.

Top-Down versus Bottom-Up Programming

A programmer is tasked with developing software that will play MP3 music files. How should the programmer begin conceptualizing the challenge of turning bits in a file into music we can hear? Should she start at the "top," thinking about how the music will sound, and how the MP3 player will look and behave? Or should she start at the "bottom," thinking about the low-level device drivers required to receive a stream of bits and convert them into audio wave forms?

Top-Down (TD) *programming* starts with the broadest and highest level requirements (the concept of the final program) and works down towards the low-level technical implementation details. *Bottom-Up programming* is the reverse: it starts with the low-level technical implementation details and works up to the concept of the complete program.

Both methods pose risks: what if the Top-Down approach made incorrect assumptions on the performance of the low-level devices? On the other hand, Bottom-Up risks wasting time by performing lots of programming for features which may not be required or implemented in the final product.

Procedural languages such as C have historically been programmed Top-Down style: start with the main program, define the procedures, and work down from there. Object-oriented programming typically uses bottom-up design: define the objects, and use them to build up to the final program.

Types of Publicly-Released Software

Once programmed, publicly-released software may come in different forms (such as with or without the accompanying source code) and released under a variety of licenses.

Open and Closed Source Software

Closed source software is software typically released in executable form: the source code is kept confidential. Examples include Oracle and Microsoft Windows 7. *Open source* software publishes source code publicly, allowing anyone to inspect, modify, or compile the code themselves. Examples include Ubuntu Linux and the Apache web server. Proprietary software is software that is subject to intellectual property protections such as patents or copyrights. "Closed source software" and "proprietary software" are sometimes used as synonyms, but that is not always true: some open source software is also proprietary.

Free Software, Shareware, and Crippleware

Free software is a controversial term that is defined differently by different groups. "Free" may mean it is free of charge to use (sometimes called "free as in beer"), or "free" may mean the user is free to use the software in any way they would like, including modifying it (sometimes called "free as in liberty"). The two types are called gratis and libre, respectively. The confusion derives from the fact that "free" carries multiple meanings in English. Software that is both gratis and libre is sometimes called free^2 (free squared).

Freeware is "free as in beer" (gratis) software, which is free of charge to use. *Shareware* is fully-functional proprietary software that may be initially used free of charge. If the user continues to use the Shareware for a specific period of time specified by the license (such as 30 days), the Shareware license typically requires payment. *Crippleware* is partially-functioning proprietary software, often with key features disabled. The user is typically required to make a payment to unlock the full functionality.

Software Licensing

Software may be released into the public domain, meaning it is (expressly) not copyrighted or licensed. This places no intellectual property constraints of the software's users. Some free (libre) software falls into this category. Most software, both closed and open source, is protected by software licensing.

Proprietary software is usually copyrighted (and possibly patented, see Chapter 11, Domain 10: Legal regulations, Investigations, and Compliance for more information on copyrights and patents); the users of the software must usually agree to the terms of the software licensing agreement before using the software. These agreements are often called *EULAs* (End-User License Agreements), which are usually agreed to when the user clicks "I agree" while installing the software.

Open source software may be protected by a variety of licensing agreements, including the GNU Public License (GPL), BSD (Berkeley Software Distribution), and Apache (named after the Apache Software Foundation) licenses.

The most prevalent of open source licenses is the GPL, which focuses on free (libre) software, allowing users the freedom to use, change, and share software. The core of the GPL is the term "copyleft," a play on copyright: copyleft seeks to ensure that free (libre) software remains free. *A Quick Guide to GPLv3* (see: http://www.gnu.org/licenses/quick-guide-gplv3.html) states: "Nobody should be restricted by the software they use. There are four freedoms that every user should have:

- The freedom to use the software for any purpose,
- The freedom to change the software to suit your needs,
- The freedom to share the software with your friends and neighbors, and
- The freedom to share the changes you make."[4]

The GPL copyleft requires modifications to GPL software to remain free: you cannot take GPL code, alter it, and make the altered code proprietary. Other free licenses, such as BSD, allow licensed code to become proprietary.

APPLICATION DEVELOPMENT METHODS

Computer programming dates to the dawn of electronic computers, in the late 1940s. Programmers first used machine code or assembly; the first high-level programming language was Fortran, which debuted in 1954. The original computer programmers often worked alone, creating entire programs as a solo effort. In that case, project management methodologies were simple or unnecessary: the programmer could sometimes conceptualize the entire project in (human) memory, and then simply write the code. As software has grown in complexity, software programming has increasingly become a team effort. Team-based projects require project management: providing a project framework with deliverables and milestones, divvying up tasks, team communication, progress evaluation and reporting, and (hopefully) a final delivered product.

Ultimately, large application development projects may closely resemble projects that have nothing to do with software, like making widgets or building bridges. Application development methods such as the Waterfall and Spiral Models are often close cousins to nonprogramming models. These methods can be thought of as project management methods, with additional features to support the creation of code.

Waterfall Model

The *Waterfall Model* is a linear application development model that uses rigid phases; when one phase ends, the next begins. The Waterfall Model predates software design and was first used in manufacturing. It was first used to describe a software development process in 1969, when large software projects had become too complex to design using informal methods. Steps occur in sequence, and the unmodified waterfall model does not allow developers to go back to previous steps. It is called the waterfall because it simulates water falling: it cannot go back up.

The Waterfall Model was first described in relation to developing software in "Managing the Development of Large Software Systems" by Dr Winston W Royce (see: http://leadinganswers.typepad.com/leading_answers/files/original_waterfall_paper_winston_royce.pdf). Royce's unmodified waterfall (with no iteration, sometimes called "stagewise") is shown in Figure 9.1, and includes the following steps: System requirements, Software Requirements, Analysis, Program Design, Coding, Testing, and Operations.

Royce's paper did not use the term "waterfall," but he described the process. An unmodified waterfall does not allow iteration: going back to previous steps. This places a heavy planning burden on the earlier steps. Also, since each subsequent step cannot begin until the previous step ends, any delays in earlier steps cascade through to the later steps.

Ironically, Royce's paper was a criticism of the model. Regarding the model shown in Figure 9.1, "the implementation described above is risky and invites failure."[5] In the real world, iteration is required: it is not (usually) realistic to prohibit a return to previous steps: Royce raised the issue of discovering a fundamental design error during the testing phase: "The testing phase which occurs at the end of the development cycle is the first event for which timing, storage, input/output transfers, etc., are experienced as distinguished from analyzed. These phenomena are not precisely analyzable . . . . Yet if these phenomena fail to satisfy the various external constraints, then invariably a major redesign is required."[5] Many subsequent software design models are called iterative models: they are explicitly designed to allow iteration: a return to previous steps.

EXAM WARNING

The specific names of the phases of Royce's unmodified Waterfall Model are not specifically testable: learn the overall flow. Also, Royce omitted a critical final step: destruction. No development process that leads to an operational system with sensitive production data is truly complete until that system has been retired, the data archived, and the remaining data on those physical systems securely destroyed.

Royce described a modified waterfall model that allowed a return to a previous phase for verification or validation, ideally confined to connecting steps. Barry Boehm's paper "A Spiral Model of Software Development and Enhancement" (see "Spiral Model" section below) shows a modified waterfall based on Royce's paper, shown in Figure 9.2.

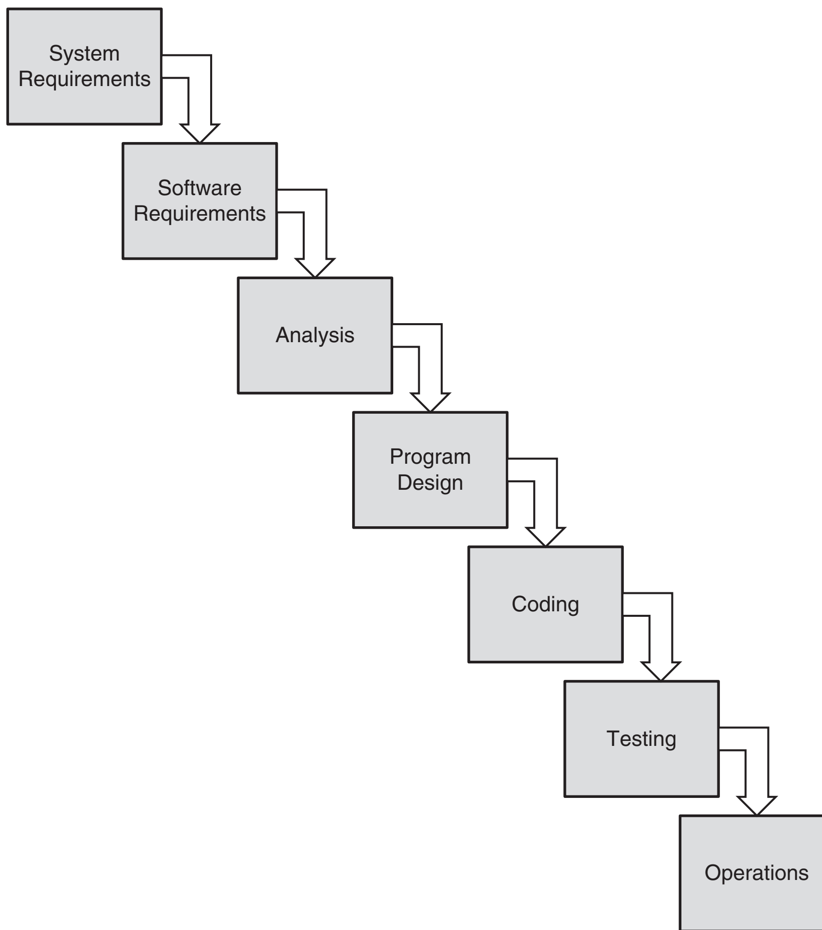

FIGURE 9.1

Unmodified Waterfall Development Model.[5]

Others have proposed similar modifications, or broadening the waterfall model. The Sashimi Model is based on (and a reaction to) the Waterfall Model.

> **NOTE**
>
> The unmodified Waterfall Model does not allow going back. The modified Waterfall Model allows going back at least one step.

Sashimi Model

The *Sashimi Model* has highly overlapping steps; it can be thought of as a real-world successor to the Waterfall Model (and is sometimes called the Sashimi Waterfall Model). It is named after the Japanese delicacy Sashimi, which has

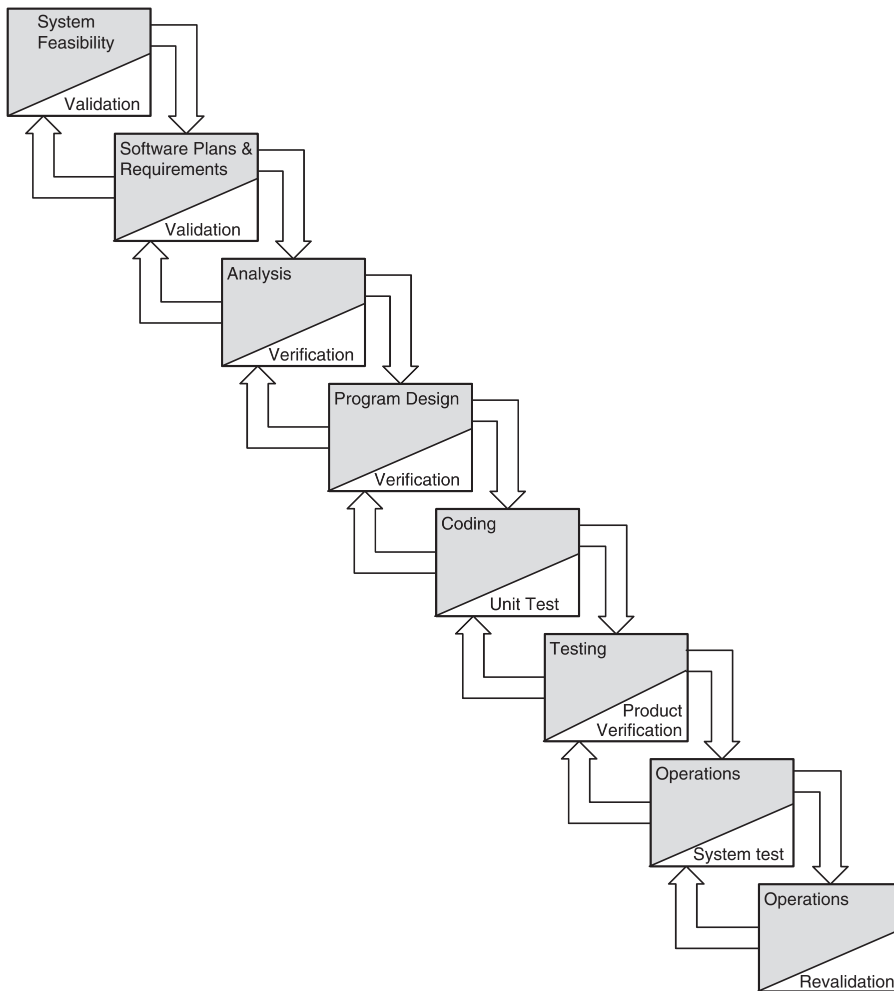

FIGURE 9.2

Modified Waterfall Development Model.[10]

overlapping layers of fish (and also a hint for the exam). The model is based on the hardware design model used by Fuji-Xerox: "Business scholars and practitioners were asking such questions as 'What are the key factors to the Japanese manufacturers' remarkable successes?' and 'What are the sources of their competitive advantage?' The sashimi system seems to give answers to these questions."[6]

Peter DeGrace described Sashimi in relation to software development in his book "Wicked problems, righteous solutions: a catalogue of modern software." Sashimi's steps are similar to the Waterfall Model's; the difference is the explicit overlapping, shown in Figure 9.3.

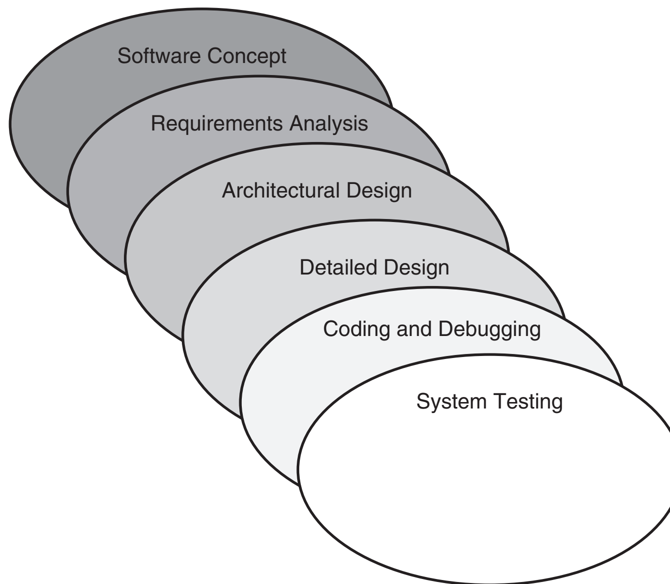

FIGURE 9.3

The Sashimi Model.[27]

Agile Software Development

Agile Software Development evolved as a reaction to rigid software development models such as the Waterfall Model. Agile methods include *Scrum* and *Extreme Programming* (XP). The Agile Manifesto (See: http://agilemanifesto.org/) states:

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan"[7]

Agile embodies many modern development concepts, including more flexibility, fast turnaround with smaller milestones, strong communication within the team, and more customer involvement.

Scrum

The Scrum development model (named after a scrum in the sport of rugby) is an Agile model first described in "The New New Product Development Game" by Hirotaka Takeuchi and Ikujiro Nonaka in relation to product development; they said "Stop running the relay race and take up rugby."[8] The "relay race" is the waterfall, where teams hand work off to other teams as steps are completed. They suggested: "Instead, a

holistic or 'rugby' approach-where a team tries to go the distance as a unit, passing the ball back and forth-may better serve today's competitive requirements."[8]

Peter DeGrace (of Sashimi fame) described (and named) Scrum in relation to software development. Scrums contain small teams of developers, called the *Scrum Team*. They are supported by a *Scrum Master*, a senior member of the organization who acts like a coach for the team. Finally, the *Product Owner* is the voice of the business unit.

Extreme Programming (XP)

Extreme Programming (XP) is an Agile development method that uses pairs of programmers who work off a detailed specification. There is a high level of customer involvement. "Extreme Programming improves a software project in five essential ways; communication, simplicity, feedback, respect, and courage. Extreme Programmers constantly communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by testing their software starting on day one. They deliver the system to the customers as early as possible and implement changes as suggested."[9]. XP core practices include:

- Planning: specifies the desired features, which are called the User Story. They are used to determine the iteration (timeline) and drive the detailed specifications.
- Paired programming: programmers work in teams.
- Forty-hour workweek: the forecasted iterations should be accurate enough to forecast how many hours will be required to complete the project. If programmers must put in additional overtime, the iteration must be flawed.
- Total customer involvement: the customer is always available, and carefully monitors the project.
- Detailed test procedures: they are called Unit Tests.[9]

> **NOTE**
>
> The XP development model is not to be confused with Microsoft Windows XP: Extreme Programming's use of the acronym "XP" predates Microsoft's use.

Spiral

The Spiral Model is a software development model designed to control risk. Barry W. Boehm created the model, described in his 1986 paper "A Spiral Model of Software Development and Enhancement" (see: http://portal.acm.org/citation.cfm?id=12948). Boehm states "The major distinguishing feature of the spiral model is that it creates a risk-driven approach to the software process rather than a primarily document-driven or code-driven process. It incorporates many of the strengths of other models and resolves many of their difficulties."[10]

The spiral model repeats steps of a project, starting with modest goals, and expanding outwards in ever wider spirals (called rounds). Each round of the spiral constitutes a project, and each round may follow traditional software development

methodology such as Modified Waterfall. A risk analysis is performed each round. Fundamental flaws in the project or process are more likely to be discovered in the earlier phases, resulting in simpler fixes. This lowers the overall risk of the project: large risks should be identified and mitigated.

Boehm used the Spiral Model to develop the TRW Software Productivity System (TRW-SPS), a complex software project that resulted in 1,300,000 computer instructions. "Round zero" was a feasibility study, a small project designed to determine if the TRW-SPS project represented significant value to the organization, and was thus worth the risk of undertaking. The feasibility study indicated that the project was worthwhile (low risk), and the project spiraled outward. The deliverables of further rounds included:

1. Concept of Operations (COOP)
2. Software Requirements
3. Software Product Design
4. Detailed Design[10]

Each round included multiple repeated steps, including prototype development and, most importantly, a risk analysis. Boehm's spiral is shown in Figure 9.4.

The spiral ended with successful implementation of the project. Any potential high risk, such as lack of value to the organization or implementation failure, was identified and mitigated earlier in the spiral, when it was cheaper and easier to mitigate.

Rapid Application Development (RAD)

Rapid Application Development (RAD) rapidly develops software via the use of prototypes, "dummy" GUIs, back-end databases, and more. The goal of RAD is quickly meeting the business need of the system; technical concerns are secondary. The customer is heavily involved in the process.

According to the Centers for Medicare & Medicaid Services (see: http://www.cms.gov/SystemLifecycleFramework/Downloads/SelectingDevelopmentApproach.pdf), RAD "Aims to produce high quality systems quickly, primarily through the use of iterative prototyping (at any stage of development), active user involvement, and computerized development tools. These tools may include Graphical User Interface (GUI) builders, Computer Aided-Software Engineering (CASE) tools, Database Management Systems (DBMS), fourth-generation programming languages, code generators, and object-oriented techniques."[11]

Prototyping

Prototyping is an iterative approach which breaks projects into smaller tasks, creating multiple mockups (prototypes) of system design features. This lowers risk by allowing the customer to see realistic-looking results long before the final product is completed. As with other modern development methods, there is a high level of customer involvement: the customer inspects the prototypes to ensure that the project is on track and meeting its objective.

FIGURE 9.4

The Spiral Model.[10]

The term "prototype" may be a bit misleading: later stage prototypes may be used as the actual final product. Prototypes can be thought of as "working model." Prototyping is not a full-fledged software development methodology: it is used by other iterative methods such as Spiral or RAD.

SDLC

The *Systems Development Life Cycle* (*SDLC*, also called the *Software Development Life Cycle* or simply the *System Life Cycle*) is a system development model. SDLC is used across the IT industry, but SDLC focuses on security when used in context of the exam. Think of "our" SDLC: as the "*Secure* Systems Development Life Cycle": the security is implied.

On the exam, SDLC focuses on security in every phase. This model is broader than many application development models, focusing on the entire system, from

selection/development, through operational requirements, to secure disposal. There are many variants of the SDLC, but most follow (or are based on) the National Institute of Standards and Technology (NIST) SDLC process.

NIST Special Publication 800-14 states: "Security, like other aspects of an IT system, is best managed if planned for throughout the IT system life cycle. There are many models for the IT system life cycle but most contain five basic phases: initiation, development/acquisition, implementation, operation, and disposal."[12] Additional steps are often added, most critically the security plan, which is the first step of any SDLC. The following overview is summarized from NIST SP 800-14:

- Prepare a Security Plan: Ensure that security is considered during all phases of the IT system life cycle, and that security activities are accomplished during each of the phases.
- Initiation: The need for a system is expressed and the purpose of the system is documented.
  - Conduct a Sensitivity Assessment: Look at the security sensitivity of the system and the information to be processed.
- Development/acquisition: The system is designed, purchased, programmed or developed.
  - Determine Security Requirements: Determine technical features (like access controls), assurances (like background checks for system developers), or operational practices (like awareness and training).
  - Incorporate Security Requirements Into Specifications: Ensure that the previously gathered information is incorporated in the project plan.
  - Obtain the System and Related Security Activities: May include developing the system's security features, monitoring the development process itself for security problems, responding to changes, and monitoring threats.
- Implementation: The system is tested and installed.
  - Install/Turn-On Controls: A system often comes with security features disabled. These need to be enabled and configured.
  - Security Testing: Used to certify a system; may include testing security management, physical facilities, personnel, procedures, the use of commercial or in-house services (such as networking services), and contingency planning.
  - Accreditation: The formal authorization by the accrediting (management) official for system operation and an explicit acceptance of risk.
- Operation/Maintenance: The system is modified by the addition of hardware and software and by other events.
  - Security Operations and Administration: Examples include backups, training, managing cryptographic keys, user administration, and patching.
  - Operational Assurance: Examines whether a system is operated according to its current security requirements.
  - Audits and Monitoring: A system audit is a one-time or periodic event to evaluate security. Monitoring refers to an ongoing activity that examines either the system or the users.

- Disposal: The secure decommission of a system.
  - Information: Information may be moved to another system, archived, discarded, or destroyed.
  - Media Sanitization: There are three general methods of purging media: overwriting, degaussing (for magnetic media only), and destruction.[12]

Notice that the word "secure" or "security" appears somewhere in every step of NIST's SDLC, from project initiation to disposal: this is the crux of the SDLC.

> **NOTE**
>
> Security is part of every step of "secure" SDLC on the exam. Any step that omits security is the "wrong answer." Also, any SDLC plan that omits secure disposal as the final lifecycle step is also the "wrong answer."

Many organizations have broadened the SDLC process, beginning with the framework described in NIST SP 800-14, and adding more steps. The United States Department of Justice (DOJ) describes a 10-step SDLC (see: http://www.justice.gov/jmd/irm/lifecycle/ch1.htm).The text from the DOJ SDLC graphic, shown in Figure 9.5, is summarized here:

> **EXAM WARNING**
>
> Memorizing the specific steps of each SDLC is not required, but be sure to understand the logical (secure) flow of the SDLC process.

- "Initiation: Begins when a sponsor identifies a need or an opportunity. Concept Proposal is created
- System Concept Development: Defines the scope or boundary of the concept. Includes Systems Boundary Document, Cost Benefit Analysis, Risk Management Plan and Feasibility Study
- Planning: Develops a Project Management Plan and other planning documents. Provides the basis for acquiring the resources needed to achieve a solution
- Requirements Analysis: Analyzes user needs and develops user requirements. Creates a detailed Functional Requirements Document
- Design: Transforms detailed requirements into complete, detailed System Design Document. Focuses on how to deliver the required functionality
- Development: Converts a design into a complete information system. Includes acquiring and installing systems environment; creating and testing databases/preparing test case procedures; preparing test files; coding, compiling, refining programs; performing test readiness review and procurement activities
- Integration and Test: Demonstrates that the developed system conforms to requirements as specified in the Functional Requirements Document. Conducted by the Quality Assurance staff and users. Produces Test Analysis Reports

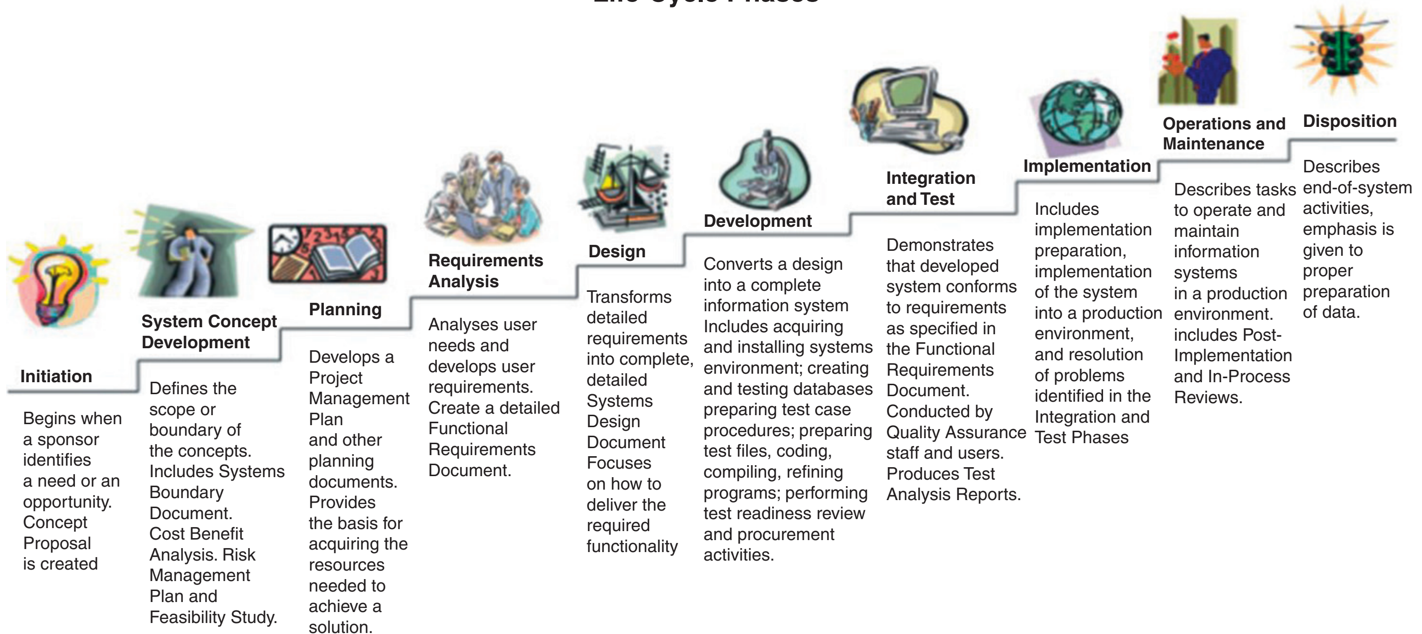

FIGURE 9.5

The DOJ SDLC.[13]

- Implementation: Includes implementation preparation, implementation of the system into a production environment, and resolution of problems identified in the Integration and Test Phase
- Operations and Maintenance: Describes tasks to operate and maintain information systems in a production environment. Includes Post-Implementation and In-Process Reviews
- Disposition: Describes end-of-system activities. Emphasis is given to proper preservation of data"[13]

Software Escrow

Software escrow describes the process of having a third party store an archive or computer software. This is often negotiated as part of a contract with a proprietary software vendor. The vendor may wish to keep the software source code secret, but the customer may be concerned that the vendor could go out of business (potentially orphaning the software). Orphaned software with no available source code will not receive future improvements or patches.

Software escrow places the source code in escrow, under the control of a neutral third party. A contract strictly specifies the conditions for potential release of the source code to the customer, typically due to the business failure of the software vendor.

OBJECT-ORIENTED DESIGN AND PROGRAMMING

Object oriented design and programming uses an object metaphor to design and write computer programs. Our bodies are comprised of objects that operate independently and communicate with each other. Our eyes are independent organs (objects) that receive input of light, and send an output of nerve impulse to our brains. Our hearts receive deoxygenated blood from our veins and oxygen from our lungs, and send oxygenated blood to our arteries. Many organs can be replaced: a diseased liver can be replaced with a healthy liver. *Object-Oriented Programming* (OOP) replicates the use of objects in computer programs. *Object-Oriented Design* (OOD) treats objects as a higher level design concept, like a flow chart.

Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) changes the older structured programming methodology, and treats a program as a series of connected objects that communicate via messages. Object-Oriented Programming attempts to model the real world. Examples of OOP languages include Java, C++, Smalltalk, and Ruby.

An object is a "black box" that is able to perform functions, and sends and receives messages. Objects contain data and *methods* (the functions they perform). The object provides *encapsulation* (also called *data hiding*): we do not know, from the outside, how the object performs its function. This provides security benefits: users should not be exposed to unnecessary details. Think of your sink as an object

whose function is washing hands. The input message is clean water; the output message is dirty water. You do not know or care about where the water is coming from, or where it is going to. If you are thinking about those issues, the sink is probably broken.

Cornerstone Object-Oriented Programming Concepts

Cornerstone object-oriented programming concepts include objects, methods, messages, inheritance, delegation, polymorphism, and polyinstantiation. We will use an example object called "Addy" to illustrate the cornerstone concepts. Addy is an object that adds two integers; it is an extremely simple object, but has enough complexity to explain core OOP concepts. Addy *inherits* an understanding of numbers and math from his *parent class* (the class is called mathematical operators). One specific object is called an *instance*. Note that objects may inherit from other objects, in addition to classes.

In our case, the programmer simply needs to program Addy to support the method of addition (inheritance takes care of everything else Addy must know). Figure 9.6 shows Addy adding two numbers.

"1 + 2" is the input message; "3" is the output message. Addy also supports delegation: if he does not know how to perform a requested function, he can delegate that request to another object (called "Subby" in Figure 9.7).

Addy also supports polymorphism (based on the Greek roots "poly" and "morph," meaning many and forms, respectively): he has the ability to overload his plus ("+") operator, performing different methods depending on the context of the input message. For example: Addy adds when the input message contains "number+number"; polymorphism allows Addy to concatenate two strings when the input message contains "string+string," as shown in Figure 9.8.

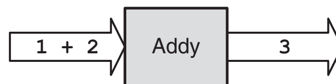

FIGURE 9.6

The "Addy" Object.

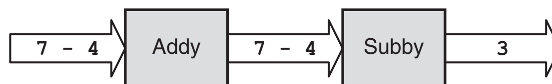

FIGURE 9.7

Delegation.

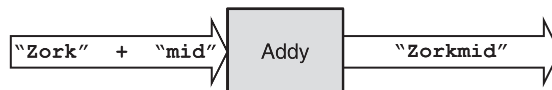

FIGURE 9.8

Polymorphism.

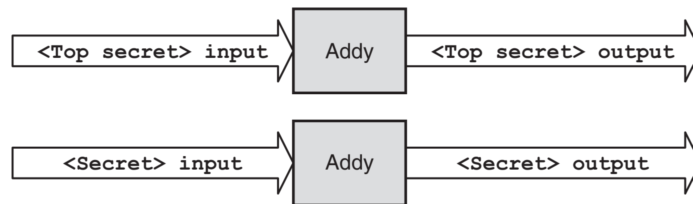

FIGURE 9.9

Polyinstantiation.

Finally, polyinstantiation means "many instances," two instances (specific objects) with the same names that contain different data. This may be used in multilevel secure environments to keep top secret and secret data separate, for example. See Chapter 6, Domain 5: Security Architecture and Design for more information about polyinstantiation. Figure 9.9 shows polyinstantiated Addy objects: two objects with the same name but different data. Note that these are two separate objects. Also, to a secret-cleared subject, the Addy object with secret data is the only known Addy object.

Here is a summary of Object-Oriented Programming concepts illustrated by Addy:

- Object: Addy
- Class: Mathematical operators
- Method: Addition
- Inheritance: Addy inherits an understanding of numbers and math from his parent class mathematical operators. The programmer simply needs to program Addy to support the method of addition
- Example input message: 1 + 2
- Example output Message: 3
- Polymorphism: Addy can change behavior based on the context of the input, overloading the "+" to perform addition, or concatenation, depending on the context
- Polyinstantiation: Two Addy objects (secret and top secret), with different data

Coupling and Cohesion

Coupling and cohesion are two concepts used to describe objects. A highly coupled object (such as Addy) requires lots of other objects to perform basic jobs, like math. An object with high cohesion is far more independent: it can perform most functions independently. Objects with high coupling have low cohesion, and the reverse is also true: objects with low coupling have high cohesion.

Addy is highly coupled and has low cohesion: he must delegate any message that does not contain a "+." Imagine another object called "Calculator," which can add, subtract, multiply, divide, perform square roots, exponentiation, etc. Calculator would have high cohesion and low coupling.

LEARN BY EXAMPLE: *MANAGING RISK THOUGH OBJECTS*

Objects are designed to be reused: this lowers development costs. Objects can also lower risk. Much like strong encryption such as AES, the longer an object remains in secure use, the more assurance we have that the object is truly secure. Like encryption algorithms, as time passes, and countless attacks prove unsuccessful, the object demonstrates its real-world strength.

Let us assume your company has been selling information security books online for the past 5 years. Your website allows users to choose a book, such as *TCP/IP Illustrated* by W. Richard Stevens, and enter their name, address, and credit card billing information. Credit card transactions are risky: risks include disclosure of customer's PII, as well as risk of credit card fraud: stolen cards used to fraudulently purchase books.

The website is programmed in an object-oriented language. It includes a credit card processing object called CCValidate, first written 5 years ago. The input message is the credit card number and expiration date entered by the customer. The output message is binary: "approved" or "denied."

The CCValidate object hides the complexity of what is happening in the background after the input message of credit card number and expiration date are entered. It performs the following methods:

1. The object has variable buffers for the credit card number that perform bounds checking.
2. The object ensures that the input message is the proper length and contains the proper types of characters in each field.
   a. In the case of a Master Card, 16 numbers (the credit card number), followed by the date (two-digit month followed by a four-digit year).
   b. Any input message that does not meet these criteria is immediately rejected.
3. The object ensures the expiration date is in the future.
   a. Any input message that does not meet this criteria is immediately rejected.
4. The object then evaluates the format and self-checking digits within the entered credit card number.
   a. Valid Master Card numbers start with 51-55, and have 16 digits.
   b. They must also contain proper self-checking digits.
      i. See: http://www.beachnet.com/~hstiles/cardtype.html for more information
   c. Any input message that does not meet these criteria is immediately rejected.
5. The object then sends a message to the proper credit card company server, checking to see if the card is valid and contains enough balance to make a purchase.
   a. The credit card company sends a return message of "accept" or "denied," which the credit card object sends to the web server as a message.

As CCValidate is used, bugs may be discovered and fixed. Improvements may be identified and coded. Over time, the object matures and simply does its job. It is attacked on the Internet; attackers launch buffer overflow attacks and insert garbage numbers, and the object performs admirably.

If a new site comes online, the programmers should not create a new credit card validating object by scratch: reinventing the wheel is too risky. They should manage their risk by locating and using a mature object that has stood the test of time: CCValidate.

Object Request Brokers

As we have seen previously, mature objects are designed to be reused: they lower risk and development costs. *Object Request Brokers* (ORBs) can be used to locate objects: they act as object search engines. ORBs are *middleware*: they connect programs to programs. Common object brokers included COM, DCOM, and CORBA.

COM and DCOM

Two object broker technologies by Microsoft are *COM* (*Component Object Model*) and *DCOM* (*Distributed Component Object Model*). COM locates objects on a local system; DCOM can also locate objects over a network.

COM allows objects written with different OOP languages to communicate, where objects written in C++ send messages to objects written in Java, for example. It is designed to hide the details of any individual object, and focuses on the object's capabilities. According to Microsoft (see: http://www.microsoft.com/com/default.mspx), COM "is used by developers to create reusable software components, link components together to build applications, and take advantage of Windows services. COM objects can be created with a variety of programming languages. Object-oriented languages, such as C++, provide programming mechanisms that simplify the implementation of COM objects. The family of COM technologies includes COM+, Distributed COM (DCOM), and ActiveX® Controls."[14] COM+ is an extension to COM, introduced in Microsoft Windows 2000. ActiveX is discussed in Chapter 6, Domain 5: Security Architecture and Design.

DCOM is a networked sequel to COM: "Microsoft® Distributed COM (DCOM) extends the Component Object Model (COM) to support communication among objects on different computers—on a LAN, a WAN, or even the Internet. With DCOM, your application can be distributed at locations that make the most sense to your customer and to the application."[15] DCOM includes *Object Linking and Embedding* (OLE), a way to link documents to other documents.

Both COM and DCOM are being supplanted by Microsoft.NET, which can interoperate with DCOM, but offers advanced functionality to both COM and DCOM.

CORBA

Common Object Request Broker Architecture (CORBA) is an open vendor-neutral networked object broker framework by the Object Management Group (OMG). CORBA competes with Microsoft's proprietary DCOM. CORBA objects communicate via a message interface, described by the *Interface Definition Language* (IDL). See http://www.corba.org for more information about CORBA.

The essence of CORBA, beyond being a networked object broker, is the separation of the interface (syntax for communicating with an object) from the instance (the specific object): "The interface to each object is defined very strictly. In contrast, the implementation of an object—its running code, and its data—is hidden from the rest of the system (i.e., encapsulated) behind a boundary that the client may not cross. Clients access objects only through their advertised interface, invoking only those operations that that the object exposes through its IDL interface, with only those parameters (input and output) that are included in the invocation."[16]

In addition to locating objects over a network, CORBA enforces fundamental object-oriented design: low-level details are encapsulated (hidden) from the client. The objects perform their methods without revealing how they do it. Implementers focus on connections, and not on code.

Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD)

Object-Oriented Analysis (OOA) and *Object-Oriented Design* (OOD) are a software design methodology that takes the concept of objects to a higher, more conceptual, level than OOP. The two terms are sometimes combined as Object-Oriented Analysis and Design (OOAD).

It is like drawing a flowchart on a whiteboard which shows how a program should conceptually operate. The way data in a program flows and is manipulated is visualized as a series of messages and objects. Once the software design is complete, the code may be programmed in an OOP language such as Ruby.

Object-Oriented Analysis (OOA) seeks to understand (analyze) a *problem domain* (the challenge you are trying to address) and identifies all objects and their interaction. Object-Oriented Design (OOD) then develops (designs) the solution.

We will use Object-Oriented Analysis and Design to design a networked intrusion detection system (NIDS). As we learned in Chapter 8, Domain 7: Telecommunications and Network Security, an NIDS performs the following actions:

1. Sniffs packets from a network and converts them into pcap (packet capture) format;
2. Analyzes the packets for signs of attacks, which could include Denial of Service, client-side attacks, server-side attacks, web application attacks, and others;
3. If a malicious attack is found, the NIDS sends an alert. NIDS may send alerts via email, paging, syslog, or security information and event managers (SIEMs).

The previous steps serve as the basis for our Object-Oriented Analysis. A sniffer object receives messages from the network in the form of packets. The sniffer converts the packets to pcap (packet capture) data, which it sends to the analysis object. The analysis object performs a number of functions (methods), including detecting denial of service, client-side, server-side, or web application attacks. If any are detected, it sends an alert message to the alerting object. The alerting object may also perform a number of functions, including alerting via email, paging, syslog, or SIEM. The NIDS Object-Oriented Design is shown in Figure 9.10.

This NIDS design addresses the problem domain of alerting when malicious traffic is sent on the network.

SOFTWARE VULNERABILITIES, TESTING, AND ASSURANCE

Once the project is underway and software has been programmed, the next steps are testing the software, focusing on the confidentiality, integrity, and availability of the system, the application, and the data processed by the application. Special care must be given to the discovery of software vulnerabilities which could lead to data or system compromise. Finally, organizations need to be able to gauge the effectiveness of their software creation process, and identify ways to improve it.

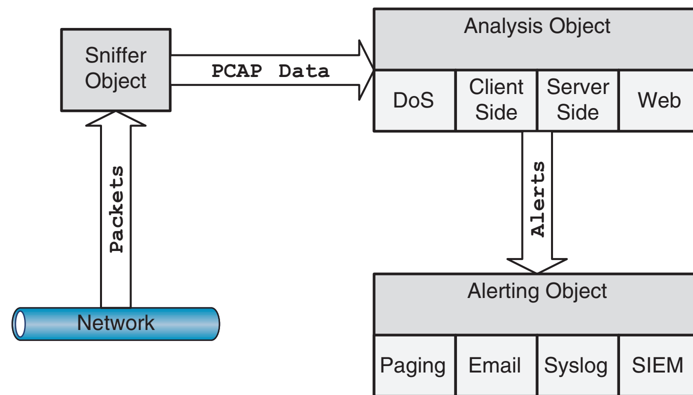

FIGURE 9.10

OOD NIDS Design.

Software Vulnerabilities

Programmers make mistakes: this has been true since the advent of computer programming. In *Code Complete*, Steve McConnell says "experience suggests that there are 15-50 errors per 1000 lines of delivered code."[17] One thousand lines of code is sometimes called a KLOC; "K" stands for thousand. This number can be lowered by following a formal application maturity framework model. Watts S. Humphrey, a Fellow at Carnegie Mellon University's Software Engineering Institute, claims that organizations that follow SEI Capability Maturity Model (CMM, see "Software Capability Maturity Model" section below) can lower the number of errors to one in every KLOC.[18]

Even 1 error per thousand lines of code can introduce large security risks, as our software becomes increasingly complex. Take Microsoft Windows, for example: "As a result, each new version of Windows carries the baggage of its past. As Windows has grown, the technical challenge has become increasingly daunting. Several thousand engineers have labored to build and test Windows Vista, a sprawling, complex software construction project with 50 million lines of code, or more than 40% larger than Windows XP."[19]

If the Microsoft Vista programmers made only one error per KLOC, then Vista has 50,000 errors. Large software projects highlight the need for robust and methodical software testing methodologies.

Types of Software Vulnerabilities

This section will briefly describe common application vulnerabilities. More technical details on vulnerabilities such as buffer overflows are discussed in Chapter 6, Domain 5: Security Architecture and Design. An additional source of up-to-date vulnerabilities can be found at "2010 CWE/SANS Top 25 Most Dangerous Programming Errors," available at http://cwe.mitre.org/top25/; the following summary

is based on this list. CWE refers to Common Weakness Enumeration, a dictionary of software vulnerabilities by MITRE (see: http://cwe.mitre.org/). SANS is the SANS Institute; see http://www.sans.org.

- Hard-coded credentials: Backdoor username/passwords left by programmers in production code
- Buffer Overflow: Occurs when a programmer does not perform variable bounds checking
- SQL Injection: manipulation of a back-end SQL server via a front-end web server
- *Directory Path Traversal*: escaping from the root of a web server (such as/var/www) into the regular file system by referencing directories such as "../.."
- PHP *Remote File Inclusion* (RFI): altering normal PHP URLs and variables such as "http://good.example.com?file=readme.txt" to include and execute remote content, such as: http://good.example.com?file=http://evil.example.com/bad.php
- *Cross-Site Scripting* (XSS): Third-party execution of web scripting languages such as Javascript within the security context of a trusted site
- *Cross-Site Request Forgery* (CSRF, or sometimes XSRF): Third-party redirect of static content within the security context of a trusted site[20]

Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) are often confused. They are both web attacks: the difference is XSS executes a script in a trusted context:

```
<script>alert("XSS Test!");</script>
```

The previous code would pop up a harmless "XSS Test!" alert. A real attack would include more javascript, often stealing cookies or authentication credentials.

CSRF often tricks a user into processing a URL (sometimes by embedding the URL in an HTML image tag) that performs a malicious act, for example tricking a white hat into rendering the following image tag:

```
<img src="https://bank.example.com/transfer-money?from=WHITEHAT&
to=BLACKHAT">
```

Software Testing Methods

There are a variety of software testing methods. In addition to the testing the features and stability of the software, testing increasingly focuses on discovering specific programmer errors that could lead to vulnerabilities which risk system compromise, including a lack-of-bounds checking.

Static testing tests the code passively: the code is not running. This includes walkthroughs, syntax checking, and code reviews. *Dynamic testing* tests the code while executing it.

White box software testing gives the tester access to program source code, data structures, variables, etc. *Black box testing* gives the tester no internal details: the software is treated as a black box that receives inputs.

A *Traceability Matrix* (sometimes called a Requirements Traceability Matrix, or RTM) can be used to map customer's requirements to the software testing plan: it "traces" the "requirements," and ensures that they are being met.

Software Testing Levels

It is usually helpful to approach the challenge of testing software from multiple angles, addressing various testing levels, from low to high. The software testing levels of Unit Testing, Installation Testing, Integration Testing, Regression Testing, and Acceptance Testing are designed to accomplish that goal:

- *Unit Testing*: Low-level tests of software components, such as functions, procedures or objects
- *Installation Testing*: Testing software as it is installed and first operated
- *Integration Testing*: Testing multiple software components as they are combined into a working system. Subsets may be tested, or *Big Bang* integration testing tests all integrated software components
- *Regression Testing*: Testing software after updates, modifications, or patches
- *Acceptance Testing*: testing to ensure the software meets the customer's operational requirements. When this testing is done directly by the customer, it is called User Acceptance Testing.

Fuzzing

Fuzzing (also called *fuzz testing*) is a type of black box testing that enters random, malformed data as inputs into software programs to determine if they will crash. A program that crashes when receiving malformed or unexpected input is likely to suffer from a boundary checking issue, and may be vulnerable to a buffer overflow attack.

Fuzzing is typically automated, repeatedly presenting random input strings as command line switches, environment variables, and program inputs. Any program that crashes or hangs has failed the fuzz test.

Combinatorial Software Testing

Combinatorial software testing is a black-box testing method that seeks to identify and test all unique combinations of software inputs. An example of combinatorial software testing is *pairwise testing* (also called *all pairs testing*).

NIST gives the following example of pairwise testing (see: http://csrc.nist.gov/groups/SNS/acts/documents/kuhn-kacker-lei-hunter09.pdf), "Suppose we want to demonstrate that a new software application works correctly on PCs that use the Windows or Linux operating systems, Intel or AMD processors, and the IPv4 or IPv6 protocols. This is a total of $2 \times 2 \times 2 = 8$ possibilities but, as (Table 9.1) shows, only four tests are required to test every component interacting with every other component at least once. In this most basic combinatorial method, known as pairwise testing, at least one of the four tests covers all possible pairs ($t = 2$) of values among the three parameters."[21]

Table 9.1 NIST Pairwise Testing Example[21]

Test case	OS	CPU	Protocol
1	Windows	Intel	IPv4
2	Windows	AMD	IPv6
3	Linux	Intel	IPv6
4	Linux	AMD	IPv4

Disclosure

Disclosure describes the actions taken by a security researcher after discovering a software vulnerability. This topic has proven controversial: what actions should you take if you discover a flaw in well-known software such as the Apache web server or Microsoft's IIS (Internet Information Services) web server?

Assuming you are a white hat (ethical) researcher, the risk is not that you understand the vulnerability: the risk is that others may independently discover the vulnerability, or may have already done so. If the others are black hats (unethical), anyone running the vulnerable software is at risk. See Chapter 3, Domain 2: Access Control for more information on types of security attackers and researches.

The ethical researcher could privately inform the vendor responsible for the software, and share the research that indicated the software was vulnerable. This process works well if the vendor quickly releases a fix or a patch for the vulnerability, but what if the vendor does nothing?

Full Disclosure is the controversial practice of releasing vulnerability details publicly. The rationale is this: if the bad guys may already have the information, then everyone should also have it. This ensures the white hats also receive the information, and will also pressure the vendor to patch the vulnerability. Advocates argue that vulnerable software should be fixed as quickly as possible; relying on (perceived) lack of knowledge of the vulnerability amounts to "Security through obscurity," which many argue is ineffective. The Full Disclosure mailing list (see: http://seclists.org/fulldisclosure/) is dedicated to the practice of full disclosure.

The practice of full disclosure is controversial (and considered unethical by many) because many black hats (including script kiddies) may benefit from this practice; zero-day exploits (exploits for vulnerabilities with no patch) are more likely to be developed, and additional innocent organizations may be harmed.

Responsible disclosure is the practice of privately sharing vulnerability information with a vendor, and withholding public release until a patch is available. This is generally considered to be the ethical disclosure option. Other options exist between full and responsible disclosure, including privately sharing vulnerability information with a vendor, but including a deadline, such as "I will post the vulnerability details publicly in three months, or after you release a patch, whichever comes first."

Software Capability Maturity Model (CMM)

The Software *Capability Maturity Model* (CMM) is a maturity framework for evaluating and improving the software development process. The model was developed by Carnegie Mellon University's (CMU) Software Engineering Institute (SEI).

The goal of CMM is to develop a methodical framework for creating quality software which allows measurable and repeatable results: "Even in undisciplined organizations, however, some individual software projects produce excellent results. When such projects succeed, it is generally through the heroic efforts of a dedicated team, rather than through repeating the proven methods of an organization with a mature software process. In the absence of an organization-wide software process, repeating results depends entirely on having the same individuals available for the next project. Success that rests solely on the availability of specific individuals provides no basis for long-term productivity and quality improvement throughout an organization. Continuous improvement can occur only through focused and sustained effort towards building a process infrastructure of effective software engineering and management practices."[22]

The five levels of CMM are described in (see: http://www.sei.cmu.edu/reports/93tr024.pdf):

(1) *Initial*: The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort.
(2) *Repeatable*: Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.
(3) *Defined*: The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. Projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.
(4) *Managed*: Detailed measures of the software process and product quality are collected, analyzed, and used to control the process. Both the software process and products are quantitatively understood and controlled.
(5) *Optimizing*: Continual process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.[22]

DATABASES

A *database* is a structured collection of related data. Databases allow queries (searches), insertions (updates), deletions, and many other functions. The database is managed by the *Database Management System* (DBMS), which controls all access to the database and enforces the database security. Databases are managed by *Database Administrators* (DBAs). Databases may be searched with a database *query language*, such as the *Structured Query Language* (SQL). Typical database security issues include the confidentiality and integrity of the stored data. Integrity is a primary concern when replicated databases are updated.

Additional database confidentiality issues include *inference* and *aggregation* attacks, discussed in detail in Chapter 6, Domain 5: Security Architecture and Design. Aggregation is a mathematical attack where an attacker aggregates details at a lower classification to determine information at a higher classification. Inference is a similar attack, but the attacker must logically deduce missing details: unlike aggregation, a mystery must be solved.

Types of Databases

Formal database types include *relational* (two dimensional), *hierarchical,* and *object-oriented*. The simplest form of database is a *flat file:* a text file that contains multiple lines of data, each in a standard format. A host file (located at/etc/hosts on UNIX systems, and c:\system32\drivers\etc\hosts on many versions of Microsoft Windows) is an example of a flat file: each entry (line) contains at least an IP address and a host name.

Relational Databases

The most common modern database is the *relational database,* which contain two-dimensional *tables* of related (hence the term "relational") data. A table is also called a relation. Tables have rows and columns: a row is a database record, called a *tuple;* a column is called an *attribute*. A single cell (intersection of a row and column) in a database is called a value. Relational databases require a unique value called the *primary key* in each tuple in a table. Table 9.2 shows a relational database employee table, sorted by the primary key (SSN, or Social Security Number).

Table 9.2 attributes are SSN, Name, and Title. Tuples include each row: 133-73-1337, 343-53-4334, etc. "Gaff" is an example of a value (cell). *Candidate keys* are any attribute (column) in the table with unique values: candidate keys in the previous table include SSN and Name; SSN was selected as the primary key because it is truly unique (two employees could have the same name, but not the same SSN). Two tables in a relational database may be joined by the primary key.

Foreign Keys

A *foreign key* is a key in a related database table that matches a primary key in the parent database. Note that the foreign key is the local table's primary key: it is called the foreign key when referring to a parent table. Table 9.3 is the HR

Table 9.2 Relational Database Employee Table

SSN	Name	Title
133-73-1337	J.F. Sebastian	Designer
343-53-4334	Eldon Tyrell	Doctor
425-22-8422	Gaff	Detective
737-54-2268	Rick Deckard	Detective
990-69-4771	Hannibal Chew	Engineer

Table 9.3 HR Database Table

SSN	Vacation Time (Days)	Sick Time (Days)
133-73-1337	15	20
343-53-4334	60	90
425-22-8422	10	15
737-54-2268	3	1
990-69-4771	15	5

database table which lists employee's vacation time (in days) and sick time (also in days); it has a foreign key of SSN. The HR database table may be joined to the parent (employee) database table by connecting the foreign key of the HR table to the primary key of the employee table.

Referential, Semantic, and Entity Integrity

Databases must ensure the integrity of the data in the tables: this is called data integrity, discussed in the "Database Integrity" section below. There are three additional specific integrity issues that must be addressed beyond the correctness of the data itself: Referential, Semantic, and Entity Integrity. These are tied closely to the logical operations of the DBMS.

Referential integrity means that every foreign key in a secondary table matches a primary key in the parent table: if this is not true, referential integrity has been broken. *Semantic integrity* means that each attribute (column) value is consistent with the attribute data type. *Entity integrity* means each tuple has a unique primary key that is not null. The HR database table shown in Table 9.3, seen previously, has referential, semantic, and entity integrity. Table 9.4, on the other hand, has multiple problems: one tuple violates referential integrity, one tuple violates semantic integrity, and the last two tuples violate entity integrity.

The tuple with the foreign key 467-51-9732 has no matching entry in the employee database table. This breaks referential integrity: there is no way to link this entry to a name or title. Cell "Nexus 6" violates semantic integrity: the sick time attribute requires values of days, and "Nexus 6" is not a valid amount of sick days. Finally, the last two tuples both have the same primary key (primary to this table; foreign key to the parent employees table); this breaks entity integrity.

Table 9.4 Database Table Lacking Integrity

SSN	Vacation Time (Days)	Sick Time (Days)
467-51-9732	7	14
737-54-2268	3	Nexus 6
133-73-1337	16	22
133-73-1337	15	20

Database Normalization

Database *normalization* seeks to make the data in a database table logically concise, organized, and consistent. Normalization removes redundant data, and improves the integrity and availability of the database. Normalization has three rules, called forms (see: http://www.informit.com/articles/article.aspx?p=30646 for more information):

- First Normal Form (1NF): Divide data into tables.
- Second Normal Form (2NF): Move data that is partially dependent on the primary key to another table. The HR Database (Table 9.3) is an example of 2NF.
- Third normal Form (3NF): Remove data that is not dependent on the primary key.[23]

Database Views

Database tables may be queried; the results of a query are called a *database view*. Views may be used to provide a *constrained user interface*: for example, nonmanagement employees can be shown their individual records only via database views. Table 9.5 shows the database view resulting from querying the employee table "Title" attribute with a string of "Detective." While employees of the HR department may be able to view the entire employee table, this view may be authorized for the captain of the detectives, for example.

The Data Dictionary

The *data dictionary* contains a description of the database tables. This is called *metadata*: data about data. The data dictionary contains database view information, information about authorized database administrator, and user accounts including their names and privileges, auditing information, among others. A critical data dictionary component is the *database schema*: it describes the attributes and values of the database tables. Table 9.6 shows a very simple data dictionary which describes the two tables we have seen previously this chapter: employees and HR.

Database Query Languages

Database query languages allow the creation of database tables, read/write access to those tables, and many other functions. Database query languages have at least two subsets of commands: *Data Definition Language* (DDL) and *Data Manipulation Language* (DML). DDL is used to create, modify, and delete tables. DML is use to query and update data stored in the tables.

Table 9.5 Employee Table Database View "Detective"

SSN	Name	Title
425-22-8422	Gaff	Detective
737-54-2268	Rick Deckard	Detective

Table 9.6 Simple Database Schema

Table	Attribute	Type	Format
Employee	SSN	Digits	###-##-####
Employee	Name	String	<30 characters>
Employee	Title	String	<30 characters>
HR	SSN	Digits	###-##-####
HR	Sick Time	Digits	### days
HR	Vacation Time	Digits	### days

The most popular relational database query language is SQL (Structured Query Language), created by IBM in 1974. Many types of SQL exist, including MySQL, PostgreSQL, PL/SQL (Procedural Language/SQL, used by Oracle), T-SQL and ANSI SQL (used by Microsoft SQL), and many others.

Common SQL commands include:

- CREATE: create a table
- SELECT: select a record
- DELETE: delete a record (or a whole table)
- INSERT: insert a record
- UPDATE: change a record

Tables are created with the CREATE command, which uses Data Definition Language to describe the format of the table that is being created. An example of a Data Manipulation Language command is SELECT, which is used to search and choose data from a table. The following SELECT command could be used to create the database view shown in Table 9.5:

```
SELECT * FROM Employees WHERE Title = "Detective"
```

This means: show any ("*") records where the Title is "Detective."

Hierarchical Databases

Hierarchical databases form a tree: the global Domain Name Service (DNS) servers form a global tree. The root name servers are at the "root zone" at the base of the tree; individual DNS entries form the leaves. www.syngress.com points to the syngress.com DNS database, which is part of the dot com (.com) top level domain (TLD), which is part of the global DNS (root zone). From the root, you may go back down another branch, down to the dot gov (.gov) TLD, to the nist.gov (National Institute of Standards and Technologies) domain, to www.nist.gov.

A special form of hierarchical database is the *network model* (referring to networks of people, not data networks): this allows branches of a hierarchical database to have two parents (two connections back to the root). Imagine an organization's org chart is stored in a database that forms a tree, with the CEO as the root of the hierarchy. In this company, the physical security staff reports to both facilities (for facility issues) and to IT (for data center physical security).

The network model allows the physical security staff to have “two bosses” in the hierarchical database: reporting through an IT manager and a facilities manager.

Object-oriented Databases

While databases traditionally contain just (passive) data, object-oriented databases combine data with functions (code) in an object-oriented framework. Object-Oriented Programming (OOP) is used to manipulate the objects (and their data), managed by an Object Database Management System (ODBMS).

Database Integrity

In addition to the previously discussed relational database integrity issues of semantic, referential, and entity integrity, databases must also ensure data integrity: the integrity of the entries in the database tables. This treats integrity as a more general issue: mitigating unauthorized modifications of data. The primary challenge associated with data integrity within a database is simultaneous attempted modifications of data. A database server typically runs multiple threads (lightweight processes), each capable of altering data. What happens if two threads attempt to alter the same record?

DBMSs may attempt to *commit* updates: make the pending changes permanent. If the commit is unsuccessful, the DBMSs can *rollback* (also called abort) and restore from a *savepoint* (clean snapshot of the database tables).

A *database journal* is a log of all database transactions. Should a database become corrupted, the database can be reverted to a back-up copy, and then subsequent transactions can be “replayed” from the journal, restoring database integrity.

Database Replication and Shadowing

Databases may be highly available (HA), replicated with multiple servers containing multiple copies of tables. Integrity is the primary concern with replicated databases: if a record is updated in one table, it must be simultaneously updated in all tables. Also, what happens if two processes attempt to update the same tuple simultaneously on two different servers? They both cannot be successful; this would violate the integrity of the tuple.

Database replication mirrors a live database, allowing simultaneous reads and writes to multiple replicated databases by clients. Replicated databases pose additional integrity challenges. A two-phase (or multiphase) commit can be used to assure integrity: before committing, the DBMS requests a vote. If the DBMSs on each server agree to commit, the changes are made permanent. If any DBMSs disagree, the vote fails, and the changes are no committed (made permanent).

A *shadow database* is similar to a replicated database, with one key difference: a shadow database mirrors all changes made to a primary database, but clients do not access the shadow. Unlike replicated databases, the shadow database is one-way (data flows from primary to shadow): it serves as a live data backup of the primary.

Data Warehousing and Data Mining

As the name implies, a *data warehouse* is a large collection of data. Modern data warehouses may store many terabytes (1000 gigabytes) or even petabytes (1000 terabytes) of data. This requires large scalable storage solutions. The storage must be high performance, and allow analysis and searches of the data.

Once data is collected in a warehouse, *data mining* is used to search for patterns. Commonly sought patterns include signs of fraud. Credit card companies manage some of the world's largest data warehouses; tracking billions of transactions per year. Fraudulent transactions are a primary concern of credit card companies that lead to millions of dollars in lost revenue. No human could possibly monitor all of those transactions, so the credit card companies use data mining to separate the signal from noise. A common data mining fraud rule monitors multiple purchases on one card in different states or countries in a short period of time. Should this occur, a violation record can be produced, leading to suspension of the card or a phone call to the card owner's home.

ARTIFICIAL INTELLIGENCE

Computers compute: they do exactly what they are told. The term "computer" was first used in 1613 to describe a person who added numbers. Artificial Intelligence is the science of programming electronic computers to "think" more intelligently, sometimes mimicking the ability of mammal brains.

Expert Systems

Expert systems consist of two main components. The first is a *knowledge base* that consists of "if/then" statements. These statements contain rules that the expert system uses to make decisions. The second component is an *inference engine* that follows the tree formed by the knowledge base, and fires a rule when there is a match.

Here is a sample "the internet is down" Expert System, which may be used by a help desk when a user calls to complain that they cannot reach the internet:

1. If your computer is turned on
   a. Else: turn your computer on
2. Then if your monitor is turned on
   a. Else: turn your monitor on
3. Then if your OS is booted and you can open a cmd.exe prompt
   a. Else: repair OS
4. Then if you can ping 127.0.0.1
   a. Else: check network interface configuration
5. Then if you can ping the local gateway
   a. Else: check local network connection

6. Then if you can ping internet address 192.0.2.187
   a. Else: check gateway connectivity
7. Then if you can ping syngress.com
   a. Else: check DNS

Forward chaining starts with no premise ("Is the computer turned on" in our previous example), and works forward to determine a solution. Backward chaining begins with a premise ("Maybe DNS is broken"), and works backwards.

The integrity of the knowledge base is critical. The entire knowledge base should form a logical tree, beginning with a trunk ("Is the computer turned on" in our previous example). The knowledge base should then branch out. The inference engine follows the tree, branching or firing as if/then statements are answered.

There should be no circular rules; an example of a circular rule using our previous example: "If your computer is turned on, then if your monitor is turned on, then if your OS is booted and you can open a cmd.exe prompt, then if your computer is turned on. . ." There should also be no unreferenced rules (branches that do not connect to the knowledge base tree).

Artificial Neural Networks

Artificial Neural Networks (ANN) simulate neural networks found in humans and animals. The human brain's neural network has 100 billion neurons, interconnected by thousands or more synapses each. Each neuron may fire based on synaptic input. This multilayer neural network is capable of making a single decision based on thousands or more inputs.

Real Neural Networks

Let us discuss how a real neural network operates: Imagine you are walking down the street in a city at night, and someone is walking behind you closely. You begin to become nervous: it is late; it is dark; and the person behind you is too close. You must make a decision: fight or flight. You must decide to turn around to face your pursuer, or to get away from them.

As you are making your decision, you weigh thousands upon thousands of inputs. You remember past experience; your instincts guide you, and you perceive the world with your five senses. These senses are sending new input to your brain, millisecond by millisecond. Your memory, instincts, sight, smell, hearing, etc., all continually send synaptic input to neurons. Less important input (such as taste in this case) has a lower synaptic weight. More important input (such as sound) has a higher synaptic weight. Neurons that receive higher input are more likely to fire, and the output neuron eventually fires (makes a decision).

Finally, you decide to turn and face your pursuer, and you are relieved to see it was a person listening to music on headphones, not paying attention to their surroundings. Thousands of inputs resulted in a binary decision: fight or flight. ANNs seek to replicate this complex decision-making process.

How Artificial Neural Networks Operate

ANNs seek to replicate the capabilities of biological neural networks. A node is used to describe an artificial neuron. Like its biologic counterpart, these nodes receive input from synapses and send output when a weight is exceeded. Single-layer ANNs have one layer of input nodes; multilayer ANNs have multiple layers of nodes, including hidden nodes, as shown in Figure 9.11. The arrows in Figure 9.11 represent the synaptic weights. Both single and multilayer artificial neural networks eventually trigger an output node to fire: this output node makes the decision.

An Artificial Neural Network learns by example via a training function: synaptic weights are changed via an iterative process, until the output node fires correctly for a given set of inputs. Artificial Neural Networks are used for "fuzzy" solutions, where exactness is not always required (or possible), such as predicting the weather.

Bayesian Filtering

Bayesian filtering is named after Thomas Bayes, an English clergyman who devised a number of probability and statistical methods including "a simple mathematical formula used for calculating conditional probabilities."[24]

Bayesian filtering is commonly used to identify spam. Paul Gram described Bayesian filtering to identify spam in his paper "A Plan for Spam" (see: www.paulgraham.com/spam.html). He described using a "corpus" of "spam" and "ham," human-selected groups of spam and nonspam, respectively. He then used Bayesian filtering techniques to automatically assign a mathematical probability that certain "tokens" (words in the email) were indications of spam.

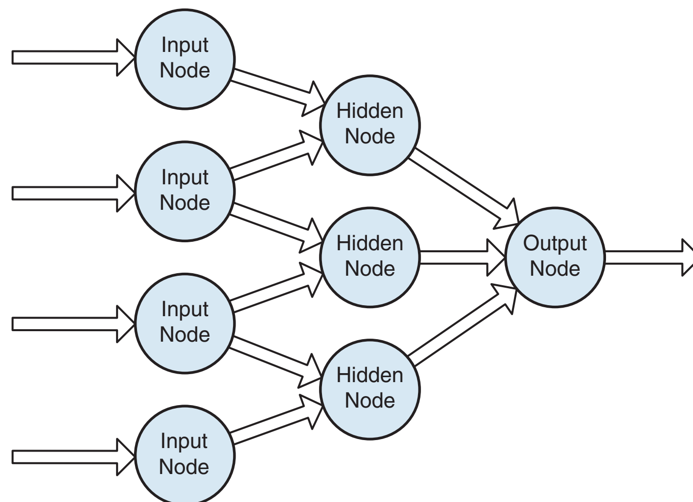

FIGURE 9.11

Multilayer Artificial Neural network.

Genetic Algorithms and Programming

Genetic Algorithms and Programming fundamentally change the way software is developed: instead of being coded by a programmer, they evolve to solve a problem. Genetic Algorithms and Programming seek to replicate nature's evolution, where animals evolve to solve problems. Genetic programming refers to creating entire software programs (usually in the form of Lisp source code); genetic algorithms refer to creating shorter pieces of code (represented as strings called chromosomes).

Both are automatically generated, and then "bred" through multiple generations to improve via Darwinian principles: "Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics. They combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search. In every generation, a new set of artificial creatures (strings) is created using bits and pieces of the fittest of the old; an occasional new part is tried for good measure. While randomized, genetic algorithms are no simple random walk. They efficiently exploit historical information to speculate on new search points with expected improved performance."[25]

Genetic programming creates random programs and assigns them a task of solving a problem. The *fitness function* describes how well they perform their task. *Crossover* "breeds" two programs together (swaps their code). *Mutation* introduces random changes in some programs. John R. Koza described the process in "Genetic Programming: On the Programming of Computers by Means of Natural Selection." The process is summarized here:

- Generate an initial population of random computer programs
- Execute each program in the population and assign it a fitness value according to how well it solves the problem.
- Create a new population of computer programs.
  - Copy the best existing programs
  - Create new computer programs by mutation.
  - Create new computer programs by crossover(sexual reproduction)[26]

Genetic Algorithms and Genetic Programming have been used to program a Pac-Man playing program, robotic soccer teams, networked intrusion detection systems, and many others.

SUMMARY OF EXAM OBJECTIVES

We live in an increasingly computerized world, and software is everywhere. The confidentiality, integrity, and availability of data processed by software are critical, as is the normal functionality (availability) of the software itself. This domain has shown how software works, and the challenges programmers face while trying to write error-free code which is able to protect data (and itself) in the face of attacks.

Following a formal methodology for developing software, followed by a rigorous testing regimen, are best practices. We have seen that following a software development maturity model such as the Capability Maturity Model (CMM) can dramatically lower the number of errors programmers make. The five steps of CMM follow the process most programming organizations follow, from an informal process to a mature process which always seeks improvement: initial, repeatable, defined, managed, and optimizing.

SELF TEST

1. What software design methodology uses paired programmers?
   - **A.** Agile
   - **B.** Extreme Programming (XP)
   - **C.** Sashimi
   - **D.** Scrum
2. What form of Artificial Intelligence uses a knowledge base and an inference engine?
   - **A.** Artificial Neural Network (ANN)
   - **B.** Bayesian Filtering
   - **C.** Expert System
   - **D.** Genetic Algorithm
3. Which of the following definitions describe open source software?
   - **A.** Freeware
   - **B.** Gnu Public License (GPL) software
   - **C.** Public domain software
   - **D.** Software released with source code
4. What type of software testing tests code passively?
   - **A.** Black box testing
   - **B.** Dynamic testing
   - **C.** Static testing
   - **D.** White box testing
5. At what phase of the (Systems Development Life Cycle) SDLC should security become part of the process?
   - **A.** Before initiation
   - **B.** During development/acquisition
   - **C.** When the system is implemented
   - **D.** SDLC does not include a security process
6. An object acts differently, depending on the context of the input message. What Object-Oriented Programming concept does this illustrate?
   - **A.** Delegation
   - **B.** Inheritance

C. Polyinstantiation
D. Polymorphism

7. Two objects with the same name have different data. What Object-Oriented Programming concept does this illustrate?
A. Delegation
B. Inheritance
C. Polyinstantiation
D. Polymorphism

8. Which software testing level tests software after updates, modifications, or patches?
A. Acceptance Testing
B. Integration Testing
C. Regression Testing
D. Unit Testing

9. What is a type of testing enters random malformed data as inputs into software programs to determine if they will crash?
A. Black box testing
B. Combinatorial testing
C. Fuzzing
D. Pairwise testing

10. What type of database language is used to create, modify, and delete tables?
A. Data Definition Language (DDL)
B. Data Manipulation Language (DML)
C. Database Management System (DBMS)
D. Structured Query Language (SQL)

11. A database contains an entry with an empty primary key. What database concept has been violated?
A. Entity Integrity
B. Normalization
C. Referential Integrity
D. Semantic Integrity

12. Which vulnerability allows a third party to redirect of static content within the security context of a trusted site?
A. Cross-Site Request Forgery (CSRF)
B. Cross-Site Scripting (XSS)
C. PHP Remote File Inclusion (RFI)
D. SQL Injection

13. What language allows CORBA (Common Object Request Broker Architecture) objects to communicate via a message interface?
A. Distributed Component Object Model (DCOM)
B. Interface Definition Language (IDL)

C. Object Linking and Embedding (OLE)
D. Object Management Guidelines (OMG)

14. What database high availability option allows multiple clients to access multiple database servers simultaneously?
A. Database commit
B. Database journal
C. Replicated database
D. Shadow database

15. What component of an expert system consists of "if/then" statements?
A. Backward chaining
B. Forward chaining
C. Inference engine
D. Knowledge base

SELF TEST QUICK ANSWER KEY

1. **B**
2. **C**
3. **D**
4. **C**
5. **A**
6. **D**
7. **C**
8. **C**
9. **C**
10. **A**
11. **A**
12. **A**
13. **B**
14. **C**
15. **D**

References

1. http://sup.netbsd.org/pub/NetBSD/NetBSD-release-4-0/src/games/trek/ram.c [accessed April 19, 2010].
2. http://snippets.dzone.com/posts/show/1885 [accessed April 19, 2010].
3. http://www.gnu.org/licenses/quick-guide-gplv3.html [accessed April 19, 2010].
4. http://www.cms.gov/SystemLifecycleFramework/Downloads/SelectingDevelopmentApproach.pdf [accessed April 19, 2010].
5. http://leadinganswers.typepad.com/leading_answers/files/original_waterfall_paper_winston_royce.pdf [accessed April 19, 2010].

6. http://www.jaist.ac.jp/ks/labs/umemoto/Fuji-Xerox.pdf [accessed April 19, 2010].
7. http://agilemanifesto.org/ [accessed April 19, 2010].
8. https://www.iei.liu.se/fek/frist/723g18/articles_and_papers/1.107457/TakeuchiNonaka1986HBR.pdf [accessed April 19, 2010].
9. http://www.extremeprogramming.org/rules.html [accessed April 19, 2010].
10. http://portal.acm.org/citation.cfm?id=12948 [accessed April 19, 2010].
11. http://csse.usc.edu/csse/TECHRPTS/2000/usccse2000-504/usccse2000-504.pdf [accessed April 19, 2010].
12. http://csrc.nist.gov/publications/nistpubs/800-14/800-14.pdf [accessed April 19, 2010].
13. http://www.justice.gov/jmd/irm/lifecycle/ch1.htm [accessed April 19, 2010].
14. http://www.microsoft.com/com/default.mspx [accessed April 19, 2010].
15. http://msdn.microsoft.com/en-us/library/ms809340.aspx [accessed April 19, 2010].
16. http://www.omg.org/gettingstarted/corbafaq.htm [accessed April 19, 2010].
17. http://www.businessweek.com/magazine/content/05_19/b3932038_mz009.htm [accessed April 19, 2010].
18. McConnell S. *Code complete: a practical handbook of software construction*. Redmond, Washington, USA: Microsoft Press; 1993.
19. http://www.nytimes.com/2006/03/27/technology/27soft.html [accessed April 19, 2010].
20. http://cwe.mitre.org/top25/ [accessed April 19, 2010].
21. http://csrc.nist.gov/groups/SNS/acts/documents/kuhn-kacker-lei-hunter09.pdf [accessed April 19, 2010].
22. http://www.sei.cmu.edu/reports/93tr024.pdf [accessed April 19, 2010].
23. http://www.informit.com/articles/article.aspx?p=30646 [accessed April 19, 2010].
24. Joyce J. Zalta EN, editor. Bayes' theorem, the stanford encyclopedia of philosophy (summer 2007 edition). 2007. http://plato.stanford.edu/archives/sum2007/entries/bayes-theorem/ [accessed April 19, 2010].
25. Goldberg DE. Genetic algorithms in search, optimization, and machine learning. Boston, MA, USA: Addison-Wesley; 1989.
26. http://www.geneticprogramming.com/Tutorial/ [accessed April 19, 2010].
27. http://www.acidaes.com/SWM.htm [accessed April 19, 2010].