CHAPTER 1

THE CASE FOR AUTOMATED DEFECT PREVENTION

COPYRIGHTED MATERIAL

> Why do we never have time to do it right, but always have time to do it over?
>
> —Anonymous

1.1 WHAT IS ADP?

ADP is a paradigm shift and a mindset. It is an approach to software development and management influenced by three distinct yet related factors:

1. The need for new and effective methodologies focusing on improving product quality
2. The fact that in today's complex world of *perpetual change*, sophisticated technology that assists software development must be an intrinsic part of project and process management, not just an add-on feature
3. An understanding of the broad spectrum of human factors affecting modern software development, in particular the psychology of learning

ADP principles and practices are based on research combined with 20 years of experience managing internal software projects, working with thousands of customers, and dealing with software defects on a daily basis. ADP evolved

Automated Defect Prevention: Best Practices in Software Management, by Dorota Huizinga and Adam Kolawa
Copyright © 2007 John Wiley & Sons, Inc.

from the approach called Automated Error Prevention (AEP) [1], used and practiced by Parasoft Corporation. Both adaptable and flexible, ADP can be applied to either existing or new projects, and it can be introduced as an extension to any iterative and incremental software process model. *When used with a new project, ADP provides a best-practice guide to defining a software development process and managing a project.*

Software has become one of the most pervasive components of our lives. Our unprecedented dependence on it ranges from keeping track of our calendars and financial records to controlling electronic devices in our automobiles, pacemakers, and a host of other applications. Yet few other goods are delivered to market with more defects than software products. This is because there are now more opportunities than ever for defects to be injected into software under development. For example, a typical enterprise system nowadays encompasses many complex multitier applications and is often a precarious combination of old and new technologies, such as legacy systems wrapped as web services and integrated with newer components through a service-oriented architecture. At each layer there are possibilities for making mistakes, and a simple defect in one component can ripple throughout the system, causing far-reaching and difficult-to-diagnose problems. Additionally, today's most common method of verifying system quality is through testing at the end of the life cycle. Unfortunately, this "quality through testing" approach is not only resource-intensive, but also largely ineffective. Since most of the time the number of possible states to be tested is prohibitively large [2], testing often leaves many system states untested, waiting only to reveal previously undetected defects at the most unexpected moment.

Thus, ADP takes an alternative approach of *comprehensive defect prevention* by modifying the development process in the entire software life cycle [3] to reduce opportunities for mistakes. In essence, ADP helps development teams prevent software faults by learning from their own mistakes and the mistakes of others. In order to achieve this, ADP describes a blueprint for life cycle defect prevention in its set of principles, practices, and policies.

At the heart of ADP lies its infrastructure, which defines the roles of people, required technology, and interactions of people with technology. This infrastructure facilitates both the implementation and sustainability of ADP processes through automation of repetitive, error-prone tasks and by automatic verification of error-preventive practices. This infrastructure also assists in the seamless collection of project-related data that is used for making informed management decisions. Thus, in ADP, the technology infrastructure, together with policies guiding its use, becomes an *intrinsic* part of project and process management.

However, no management approach can be effective unless it is based on an understanding of human nature, and aims at creating an environment that provides job satisfaction. This is particularly important in software development, an intellectually challenging task in itself, that is complicated by seemingly endless industry change that requires constant learning. ADP's

automation of tedious, repetitive, and mundane tasks combined with gradual, step-by-step introduction of new practices is an attempt to stimulate effective learning and perhaps even help achieve a highly increased sense of satisfaction by entering a peak of mental concentration called “flow” [4].

In the next section of this chapter, we will describe the goals that we set forth for ADP. This will be followed by a high-level overview of ADP’s principles, practices, and policies. The last section will delineate the relationship between ADP and modern software development.

1.2 WHAT ARE THE GOALS OF ADP?

The development of ADP was triggered by the need for effective methodologies that counter poor software quality, with its resulting high costs and operational inefficiencies. However, the high complexity of modern software development coupled with continuous changes of technology and short time to market pose a set of unique challenges not found in other industries. To address these challenges, we have defined and addressed the goals for each category of the software project management [5] spectrum, concentrating not only on the four *P*s suggested by Pressman [6]—people, product, process, and project—but on the organization as an entirety.

In subsequent sections, we will explain the primary ADP goals for each of the above categories and the motivation for each goal. (See Figure 1.1.)

1.2.1 People: Stimulated and Satisfied

People are the most important resource in an organization, as they are the sole source of creativity and intellectual power. As much as we strive to define processes and methods to be people independent, people will either make or break them.

Satisfied and motivated people are productive and cooperative. They take pride in their work and they are willing to go the extra mile to deliver a quality product. Therefore, software development, which is a people-intensive process by itself, cannot be successful without creative and dedicated people. However, professional satisfaction is not easily achieved, especially in a business where

Figure 1.1 Resources transform into goals by using ADP.

continued learning is as important as performing routine tasks and frustration can easily inhibit imagination. Moreover, achieving a balance between discipline and creativity is difficult because according to the laws of human psychology [4], in their professional lives, people tend to oscillate between two extreme states: routine and repetitive tasks on the verge of boredom, and new, challenging tasks on the verge of anxiety. *Both excessive boredom and excessive anxiety make people ineffective and error-prone.*

Part of the continuum between these two extreme states of mind includes the *competency zone*, which is the zone where people's skills match the demands of the tasks they must perform. At the high end of the competency zone is the state of *flow*. In this state people forgo their inhibitions, learn, and explore their new skills, and through a high degree of concentration, their performance is enhanced enormously, resulting in an increased level of competence. People who achieve this state report a tremendous sense of accomplishment and success.

According to Phillip G. Armour, software development is subject to the laws of flow, because it is a process of continuous learning. "If software development were entirely the application of existing knowledge, it would be a manufacturing activity and we could completely automate it" [7]. Moreover, since most software defects can ultimately be traced back to "human error," any effective defect prevention approach must create a working environment in which the team members can perform most of their tasks within the higher ends of their *competency zone*, *where the number of boring or overwhelmingly challenging activities are minimized.*

Thus, the goal of ADP is to keep people positively stimulated and yet not overwhelmed, so they can perform in an advanced manner and consequently achieve the maximum level of professional satisfaction.

1.2.2 Product: High Quality

The high quality of a product not only provides customer satisfaction and helps to maintain the company's competitive edge, but also generates a sense of individual and organizational pride among those who contributed to its development.

Software quality is a complex mix of many attributes such as usability, reliability, efficiency, and more. Focusing on just one of these factors in the development process may impede the others and undermine the ultimate measure of software quality, which is customer satisfaction. While the defect rate is one of many factors used to determine software quality, it is so fundamental that unless its status is acceptable, other aspects of quality are not as significant. Unfortunately, many past and recent reports of system failures due to software faults indicate that defects are the norm rather than the exception in software products. They cause financial losses [8,9], everyday inconvenience [10,11], and even cost lives [12,13]. A comprehensive study conducted in 2002 by the NIST (National Institute of Standards and Testing) states that software errors cost the U.S. economy up to a staggering $59.5 billion per year [14].

As previously mentioned, one of the primary contributing factors to poor software quality is its growing complexity. Multitier and multiplatform environments, unmanageable sizes reaching millions of lines of code, creeping requirements, and ever-changing underlying technology open the door for a host of defects.

Unfortunately, not many people in the industry believe that *defect prevention* in software is possible. The common claim is that because each piece of software is different, the lessons learned from working on one project cannot be effectively applied to others. Thus, instead of trying to prevent defects from entering software, the conventional approach is to test defects out of software. First, a product or its part is built, and then an attempt is made to use testing to determine whether it works. Finally, defects exposed in the testing process are gradually removed.

Yet defect prevention is not only possible, but also necessary in software development. However, for defect prevention to be effective, a formalized process for integrating this strategy into the software life cycle is needed. This formalized approach must include both the application of industry best practices known to avert common problems, and the customization of organization-specific practices to meet project needs. Additionally, in order to be sustainable, this formalized approach must be supported by an adaptable infrastructure that automates many repetitive tasks and that carries defect prevention practices from one product release to the next, and from one project to another. Moreover, the role of testing should not be eliminated, but redefined. Although back-end testing has proven to be an ineffective method of building quality into software, testing can and should be used to help measure and verify product quality. ADP defines such a formalized approach to defect prevention with the ultimate goal of achieving high quality of the product.

1.2.3 Organization: Increased Productivity and Operational Efficiency

Companies are constantly rethinking how to maintain their competitive edge by reducing operating and maintenance costs while attempting to deliver increased value. In the software industry, this manifests itself through the following goals shared by many organizations:

- *Cost reduction*: controlling the spiraling software development and labor costs, producing more with the same resources, and reducing the amount of rework due to poor quality
- *On-time product delivery*: ensuring that projects deliver products on time with the requested functionality

The inability to make effective software without incurring unreasonable costs and delivery delays is blamed on operational inefficiency with its resulting low productivity. The fact that this inability often persists in the face of increasing software development team expertise and resources indicates a

serious process problem that has little to do with insufficient resource allocation.

Unfortunately, it is often not realized that the operational inefficiency of organizations stems from the fact that in virtually any software development 80% of the resources are dedicated to identifying and fixing defects, which leaves only about 20% of the resources available for tasks that deliver value and improve the business [14].

These defects span a wide spectrum from incorrectly implemented functionality through performance problems and security vulnerabilities, to failures that crash an entire system. They essentially stifle a team's ability to produce working software within a reasonable time and at acceptable costs.

These problems, coupled with the fact that the cost of identifying and removing defects grows exponentially as a function of time in the development cycle [15], lead to the conclusion that defect prevention is crucial to improving productivity and operational effectiveness.

1.2.4 Process: Controlled, Improved, and Sustainable

A process is a series of step-by-step tasks necessary to reach a specified goal. Thus, depending on its goal, a process can be defined at different levels of granularity. A complete software development cycle needs a process, and so does each of its individual phases including requirements gathering, design, and testing. While the ultimate goal is to create a high-quality product in a timely manner, it is necessary to divide and refine each high-level goal into many subgoals for which detailed step-by-step action plans have to be prepared. For example, a software development life cycle process could consist of a requirements specification process, design process, testing process, and deployment process. However, implementation of a well-defined process is only the first step toward software product quality. The fundamental problem lies in whether and how this process can be *controlled, sustained, and improved.*

Quality initiatives, such as CMMI (Capability Maturity Model Integration) [16], which set a framework for process improvement, do not provide sufficiently practical and detailed guidelines to translate their models into actions effectively. Thus, many organizations failed to achieve the desired results from these initiatives because of the difficulty of implementing and maintaining them in realistic cost-effective development environments.

Some of the common objections to these initiatives are:

- They add a substantial overhead, which is very costly.
- They rely too much on manual labor to set up and maintain. Because of the turnover in the workforce, it is hard to sustain such human-dependent processes.
- They are difficult to automate, but without automation, they decay and eventually become ineffective.

Thus, the goal of ADP is to address these concerns by implementing software processes that are controllable and sustainable. This is accomplished by defining a set of practices, explaining how they can be automated, and by monitoring and controlling the status of the practice implementation using the ADP infrastructure.

1.2.5 Project: Managed through Informed Decision Making

A quality product cannot be created without effective project management techniques applied throughout its development. However, while project managers and developers strive to make the software better and friendlier, the economic pressures of the industry coupled with many external factors pose a multitude of challenges.

Among the external factors are recent government regulations, which place an additional burden on software teams responsible for such tasks as maintaining financial information, protecting human resources data, securing the company's product database and Web accesses, and many more. Current legislation that affects software development includes Section 508 of the U.S. Rehabilitation Act [17], the Sarbanes-Oxley Act of 2002 [18], the Health Insurance Portability and Accountability Act (HIPAA) [19], the Gramm-Leach-Bliley Act [20], and the Family Educational Rights and Privacy Act (FERPA) [21]. For example, the Sarbanes-Oxley (SOX) Act requires that public companies implement effective internal controls over their financial reporting, have auditors verify the existence and effectiveness of these internal controls, and have executives certify that financial reports are accurate. Although SOX is financial legislation, it places a tremendous burden on the software teams of public companies because reliable financial reporting is inextricably linked to a well-controlled system environment and reliable, secure software systems.

Additionally, ensuring application security has become one of the greatest challenges in recent years. Although most organizations strive to release software with zero defects, this rarely happens in reality. While in many cases little harm comes from shipping software with a few functionality defects, security weaknesses can result in great damage. Considering that attackers proactively analyze software hoping to expose vulnerabilities that they can exploit, deploying software with even one security flaw could pose a high risk. In fact, potential intruders are usually better at uncovering security defects than testing teams themselves. As a result, a defect rate that might be acceptable for software functionality could prove dangerously high for security flaws in the same application.

Another external factor affecting management of software projects is offshore outsourcing. Because of the large return on investment that outsourcing promises, many companies elect to pursue such management strategies. However, outsourcing comes with many potential risks stemming from cultural and language barriers to legislative differences that make contractual

agreements difficult to enforce. The organization's decision makers may find themselves pondering the possible disastrous consequences of the many unknowns in outsourcing: lack of understanding of company's business, geographical distance, and communication difficulties.

In order to ameliorate project uncertainty caused by the above external factors, one of the goals of ADP is to facilitate management decision making through automated collection of data and through tracking and measurements of the trends of the project status indicators. Analysis of these indicators assists in evaluating the level of project quality, status of requirements implementation, and deployment readiness, and helps to reduce the risks and challenges posed by these and other external factors.

1.3 HOW IS ADP IMPLEMENTED?

ADP is implemented by following a set of principles, practices, and policies. The principles are high-level laws that form the basis of ADP methodology, while the policies and practices comprise low-level development and management rules and procedures at varying granularity. We will expand on each of these in subsequent sections.

1.3.1 Principles

Principles are the foundation of the ADP methodology. They are the basic laws that govern structuring and managing software projects. They correspond to ADP's goals at the highest level and they form the basis for the definition of practices and policies, which are directly applicable to software projects. (see Figure 1.2.)

There are six ADP principles, which will be explained in detail in the next chapter. Each of these principles addresses one or more of the ADP goals. For example, the principle on "incremental implementation of ADP's practices and policies" assures that the organizational change that ADP brings is introduced gradually, thereby minimizing people's unease and apprehension. The incremental, group-by-group and practice-by-practice approach to ADP implementation is an attempt to minimize possible anxiety and resentment by not overwhelming people and teams who apply it. Such a gradual introduction

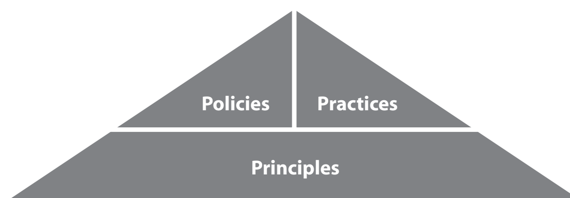

Figure 1.2 Principles, policies, and practices.

of ADP also assures that once the initial practices are mastered and accepted by one group, they can be successfully propagated to the entire organization.

1.3.2 Practices

Practices are *functional* embodiments of the principles. Depending on their level of granularity, best practices can pertain to entire projects, processes, or even individual tasks and people's daily activities. There are two types of best practices: general, which are based on issues common to all development projects, and customized, which are adopted by the organization to meet the needs of its unique projects and improve its processes. While the body of general best practices is already defined and well accepted by the industry, the body of customized best practices is created by the organization.

An example of a general best practice is managing requirements changes. This best practice would define a basic process for recording and tracking updates in software requirements. At a finer level of the granularity, this best practice would describe a specific format for recording such changes, along with the required technology and change approval process.

A customized best practice is project or organization specific. For example, a predefined set of coding standards adopted by the organization and applied by the team to a specific project is a customized best practice. Similarly, a new best practice introduced after identifying a defect in the product under development is a customized best practice used for process improvement.

DEVELOPER'S TESTIMONIAL

Customized Best Practices

At my current job many of our senior developers have worked to put together a document that has our C++ and CORBA best practices. This document helps our junior engineers to learn from the years of experience of the senior engineers. This has helped to reduce mistakes and make code easier to read and understand.

—William Mayville, Software Engineer I

1.3.3 Policies

Policies are *managerial* embodiments of the principles. They mostly pertain to teamwork and define how the team should interact with technology. They are also used to assure that product- and process-related decisions are consistently applied through the entire team, and usually take the form of written documents.

An example is a design policy for the user interface, which should define the elements in the user interface of a product and details such as each element's location, appearance, name, and functionality. Another example is a

policy for use of a requirements management system, which should define how individuals and teams use this system in order to most effectively organize and track product requirements.

1.3.4 Defect Prevention Mindset

Successful implementation of ADP practices and policies requires that at least one team member—preferably an architect, lead developer, or anyone else with a deep understanding of the software's requirements and design—assume the responsibility of identifying and removing the root causes of severe defects found. The proper mindset involves realizing that the apparent problems, such as missing requirements, failed builds, unused variables, and performance bottlenecks, are just specific symptoms of a larger, more general problem. These problems can originate anywhere in the development process, from requirements, through design and construction, to testing, and even in beta tests or user feedback. In fact, warning signs often appear downstream from the root cause, and each root cause may generate tens or hundreds of them. If each specific symptom is addressed, but not the more general and abstract root cause, the problem will persist. In the long term, it is much more effective to address the root cause, thereby preventing all related defects, than to try to tackle each one as it arises.

For example, assume that a development team member discovers that the product's automated build is not operating correctly because the wrong library was integrated into the build and old versions of functions are being called instead of the up-to-date versions. After spending significant time and effort investigating this situation, the team determines that, although the correct version of the file was stored in the source control system, an incorrect version was included in the build due to a clock synchronization problem. The build machine's clock was ahead of the source control system's clock. Consequently, the version of the file on the build machine had a more recent timestamp than the file on the source control machine, so the file on the build machine was not updated. The discrepancy in the clocks is just a symptom of the problem. Fixing the time on all of the team's computers might temporarily prevent failed file updates, but it is likely that the clocks will become unsynchronized again. The general, abstracted root cause is that there are conditions under which the most recent files from the source control system will not be transferred to the build system. This could be prevented by configuring the build process to remove all existing files on the build machine and then retrieve all of the most recent versions of the files from the source control system. Acquiring the proper mindset requires realizing that even the most seemingly insignificant symptom may result in a severe problem and point to a root cause that, if fixed, can significantly improve the process and all products affected by this process.

In this book we will give examples of how particular defects can be traced back to root problems, which can then be avoided by developing and implementing preventive action plans.

1.3.5 Automation

Automation is ADP's overarching principle and is essential to making defect prevention a sustainable strategy in software development. When key defect prevention practices are automated, organizations can ensure that these practices are implemented with minimal disruption to existing processes and projects. Moreover, automation is the solution to ensuring that both the general and customized defect prevention practices that the team decides to implement are applied thoroughly, consistently, and accurately.

In many cases, determining how to effectively automate the defect prevention strategies is just as difficult as the root cause analysis required to develop them. One of the other challenging aspects is determining how to integrate new automated practices into the development process unobtrusively so that day-to-day development activities are not disrupted unless a true problem is detected.

1.4 FROM THE WATERFALL TO MODERN SOFTWARE DEVELOPMENT PROCESS MODELS

ADP's best practice approach does not depend on any specific life cycle process model, although it is best suited for iterative and incremental development. This type of development has become prevalent in recent years because of the dynamic nature of the software industry. Due to perpetual technological changes, it is often impossible to entirely define the problem and implement the complete software solution in one cycle. Therefore, an incremental approach is taken, whereby the problem definition and solution construction undergo several iterations. Thus, modern software development has become a dynamic and living process, where modifications and reworking of project artifacts within each phase and the entire cycle are the norm.

The iterative approach, regardless of its flavor, lends itself to the application of ADP. This is because defects identified in each iteration of the life cycle or phase can be prevented from reoccurring in subsequent iterations of the same and future projects.

When defect prevention is built into the process and automated, process improvement becomes an intrinsic part of software development. This results in both a more efficient methodology and higher-quality products.

In the past decade, the software development paradigm has moved away from the traditional *waterfall approach* that features well-defined sequential stages, beginning with communication with customers and requirements specification, progressing through planning, design, construction, and deployment, and then eventually following with the maintenance of the final product. Despite its many supporters, this conventional, staged approach did not provide sufficient flexibility to accommodate the dynamic needs of today's quick-to-market business pressures, where both the technology and the

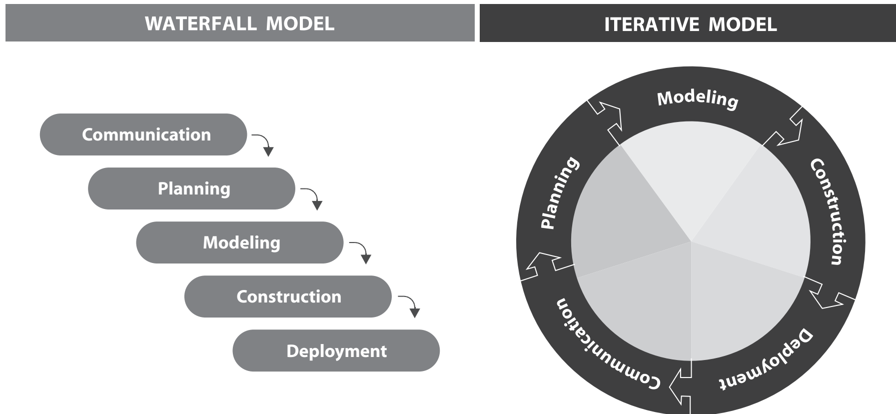

Figure 1.3 The waterfall model versus iterative process model.

customer requirements are subject to unending change. Even though the original waterfall model proposed by Winston Royce [22] suggested “feedback loops,” these were so imprecisely specified that the vast majority of the organizations applied this method in a strictly linear manner.

Consequently, this very first, classic life cycle model was replaced by the iterative process approach, whereby the initial version of the system (sometimes also called a core product) is rapidly constructed, focusing on driving requirements coupled with fundamental architecture. The software development process then undergoes a series of iterations and increments, expanding the core product until the desired levels of system functionality, performance, reliability, and other quality attributes are achieved.

Usually five generic phases are identified in the modern life cycle: communication, planning, modeling, construction, and deployment, as shown in Figure 1.3.

The principles of modern software processes focus on architecture, component-based development, automation, risk and change management, modeling, and configurable infrastructure. The architecture-first approach facilitates time and cost estimation, while the iterative life cycle makes risk control possible by gradual increases in system functionality and quality.

Software development process models define phases of software development and the sequence of their execution. They include approaches such as *incremental* [23], *spiral* [24], *object-oriented unified process* [25], *agile and extreme* [26], *and rapid prototyping and application development* [27]. Also, formal methodologies have been proposed for life cycle descriptions [28,29].

At first glance, these models might appear to be quite a departure from the traditional waterfall approach (which has well-defined sequential stages) since they blur the boundaries between development phases, often rely on close interactions with the customer, and require multiple reworking of project

artifacts within and between the development phases. Frequently customers are not capable of precisely identifying their needs early in the project, and multiple iterations of requirements definitions are essential to elicit the problem completely. Yet, a closer analysis reveals that each of these models is a natural and logical evolution of the waterfall model. In fact, these models stem from constant progress in improving the software development process. This progress is the result of efforts to improve the existing development processes in ways that would prevent the most common and disruptive problems that were causing project setbacks and product failures. Moreover, each new model still maintains the core element of the original waterfall model: a forward-moving progression through a cycle that involves requirements analysis, specification, design, implementation, testing, and maintenance. The duration, scope, and number of iterations through this cycle may vary from process to process, but its presence is essential—because it represents the natural steps of developing software. Consequently, the ability to execute the waterfall model successfully remains a requirement for success, no matter what process is used.

More discussion about software development process models is included in Appendix A.

1.5 ACRONYMS

CMMI Capability Maturity Model Integration
FERPA Family Educational Rights and Privacy Act
HIPAA Health Insurance Portability and Accountability Act
NIST National Institute of Standards and Testing
SOX Sarbanes-Oxley Act of 2002

1.6 GLOSSARY

agile programming process model A lightweight process model that consists of the following cycle: analysis of the system metaphor, design of the planning game, implementation, and integration.

extreme programming An “agile” software development methodology characterized by face-to-face collaboration between developers and an on-site customer representative, limited documentation of requirements in the form of “user stories,” and rapid and frequent delivery of small increments of useful functionality. [26]

FERPA A federal law that protects the privacy of student education records. FERPA gives parents certain rights with respect to their children’s education records. These rights transfer to the student when he or she reaches the age of 18 or attends a school beyond the high school level. [21]

Gramm-Leach-Bliley Act The Financial Modernization Act of 1999, which includes provisions to protect consumers' personal financial information held by financial institutions. [20]

HIPAA An act to amend the Internal Revenue Code of 1986 to improve portability and continuity of health insurance coverage in the group and individual markets, to combat waste, fraud, and abuse in health insurance and health care delivery, to promote the use of medical savings accounts, to improve access to long-term care services and coverage, to simplify the administration of health insurance, and for other purposes. [19]

incremental development A software development technique in which requirements definition, design, implementation, and testing occur in an overlapping, iterative (rather than sequential) manner, resulting in incremental completion of the overall software product. [23]

prototype A preliminary type, form, or instance of a system that serves as a model for later stages or for the final, complete version of that system.

Sarbanes-Oxley Act of 2002 An act to protect investors by improving the accuracy and reliability of corporate disclosures made pursuant to the securities laws, and for other purposes. [18]

Section 508 An amendment to the Rehabilitation Act of 1973 that requires that any technology produced by or for federal agencies be accessible to people with disabilities. It covers the full range of electronic and information technologies in the federal sector. [17]

software life cycle The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software life cycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and, sometimes, retirement phase. [3]

software project management The process of planning, organizing, staffing, monitoring, controlling, and leading a software project. [5]*

spiral model A model of the software development process in which the constituent activities, typically requirements analysis, preliminary and detailed design, coding, integration, and testing, are performed iteratively until the software is complete. [3]

unified process Also known as Rational Unified Process, is a software development approach that is iterative, architecture-centric, and use-case driven. [25]

usability The ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component. [3]*

use case A use case describes a sequence of actions that are performed by an actor (e.g., a person, a machine, another system) as the actor interacts with the software. An actor is a role that people or devices play as they interact with the software. Use cases help to identify the scope of the project and provide a basis for project planning. [25]

waterfall model A model of the software development process in which the constituent activities, typically a concept phase, requirements phase, design phase, implementation phase, test phase, and installation and checkout phase, are performed in that order, possibly with overlap but with little or no iteration. [3]*

* From IEEE Std. 1058.1-1987 Copyright 1987, IEEE and IEEE Std. 610.12-1990, Copyright 1990, IEEE. All rights reserved.

1.7 REFERENCES

[1] Kolawa, A., *Automated Error Prevention: Delivering Reliable and Secure Software on Time and on Budget*, 2005, http://www.parasoft.com (retrieved: July 7, 2006).

[2] Burnstein, I., *Practical Software Testing: A Process Oriented Approach*. Springer, 2002.

[3] Institute of Electrical and Electronics Engineers, *IEEE Standard 610.12-1990—Glossary of Software Engineering Terminology*, 1990.

[4] Csikszentmihalyi, Mihaly, *Flow: The Psychology of Optimal Experience*. Harper & Row, 1990.

[5] Institute of Electrical and Electronics Engineers, *IEEE Standard 1058.1-1987*, 1987.

[6] Pressman, R.S., *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005.

[7] Armour, P. G., "The Learning Edge," *Communications of ACM,* Vol. 49, No. 6, June 2006.

[8] Inquiry Board (Chairman: Prof. J.L. Lions), *ARIANE 5—Flight 501 Failure*, Paris, July 19, 1996, http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html (retrieved: July 7, 2006).

[9] National Aeronautics and Space Administration, *Mars Climate Orbiter Mishap Investigation Board Phase I Report*, 1999, http://mars.jpl.nasa.gov/msp98/news/mco991110.html (retrieved: July 7, 2006).

[10] "Maine's Medical Mistakes," *CIO*, April 15, 2006, http://www.cio.com/archive/041506/maine.html (retrieved: July 7, 2006).

[11] "2005 Toyota Prius Recalls," *AutoBuy.com*, 2005, http://www.autobuyguide.com/2005/12-aut/toyota/prius/recalls/index.html (retrieved: July 7, 2006).

[12] General Accounting Office—Information Management and Technology Division Report, *Patriot Missile Software Problem*, 1992, http://www.fas.org/spp/starwars/gao/im92026.htm (retrieved: on July 7, 2006).

[13] Leveson, N. and Turner, C.S., "An Investigation of the Therac-25 Accidents," *IEEE Computer,* Vol. 26, No. 7, July 1993, pp. 18–41.

[14] National Institute of Standards and Technology, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, Washington D.C., 2002, http://www.nist.gov/director/prog-ofc/report02-3.pdf (retrieved: July 7, 2006).

[15] Boehm, B. and Basili, B., "Software Defect Reduction Top 10 List," *IEEE Computer*, Vol. 34, No. 1, January 2001.

[16] Chrissis, M.B., Konrad, M., and Shrum, S., *CMMI—Guidelines for Process Integration and Product Improvement*, Addison Wesley, February 2005.

[17] Government Services Administration, *Summary of Section 508 Standards,* January 23, 2006, http://www.section508.gov/index.cfm?FuseAction=Content&ID=11 (retrieved: April 3, 2006).

[18] The American Institute of Certified Public Accountants, *Sarbanes-Oxley Act of 2002*, January 23, 2002, http://frwebgate.access.gpo.gov/cgibin/getdoc.cgi?dbname=107_cong_bills&docid=f:h3763enr.txt.pdf (retrieved: June 15, 2006).

[19] United States Department of Health and Human Services, *Public Law 104–191: Health Insurance Portability and Accountability Act of 1996*, August 21, 1996, http://aspe.hhs.gov/admnsimp/pl104191.htm (retrieved: July 25, 2006).

[20] Federal Trade Commission, *Privacy Initiatives: The Gramm-Leach-Bliley Act*, 1999, http://www.ftc.gov/privacy/privacyinitiatives/glbact.html (retrieved: July 25, 2006).

[21] United States Department of Education, *Family Educational Rights and Privacy Act (FERPA)*, February 17, 2005, http://www.ed.gov/policy/gen/guid/fpco/ferpa/index.html (retrieved: July 25, 2006).

[22] Royce, W.W., "Managing the Development of Large Software Systems: Concepts and Techniques," *Proceedings of IEEE WESCON*, Vol. 26, August 1970, pp. 1–9.

[23] Schach, S.R., *Object-Oriented and Classical Software Engineering*. McGraw Hill, 2002.

[24] Boehm, B.W., "A Spiral Model of Software Development and Enhancement," *IEEE Computer,* Vol. 21, No. 5, May 1988, pp. 61–72.

[25] Jacobson, I., Booch, G., and Rumbaugh, J., *The Unified Software Development Process*. Addison-Wesley, 1999.

[26] Beck, K. and Andres, C., *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2nd ed., 2004.

[27] Martin, J., *Rapid Application Development*. Prentice-Hall, 1991.

[28] Choi, J.S. and Scacchi, W., "E3SD: An Environment Supporting the Structural Correctness of SLC Descriptions," *Proceedings of the IASTED—International Conference on Software Engineering and Applications,* Las Vegas, Nevada, USA, Nov. 6–9, 2000, pp. 80–85.

[29] Choi., J.S. and Scacchi, W., "Formal Analysis of the Structural Correctness of SLC Descriptions," *International Journal of Computers and Applications,* Vol. 25, No. 2, 2003, pp: 91–97.

1.8 EXERCISES

1. What factors have influenced the development of ADP?
2. What are the goals of ADP?
3. Why is understanding of human nature, especially psychology of learning, essential in software development?
4. In what sense does psychology of "flow" apply to software development?

5. Why is it difficult to control modern processes?
6. Give examples of recent software "disasters" not listed in the book and explain their causes.
7. Give examples of recent legislation not listed in the book that might affect the IT industry and explain what kind of effect they might have.
8. What are the primary differences between ADP principles, practices, and policies?
9. Why is modern software iterative and incremental?
10. What are the key lessons to be learned from the past 35 years of software development?