Chapter 1

Overview of Testing

LEARNING OBJECTIVES

- to identify the basic mindset of a tester, regardless of what is being tested
- to determine the correct motivations for testing in business
- to explain some of the reasons why testing is undervalued as a business practice
- to explain what differentiates software testers from software developers

1.1 INTRODUCTION

There were numerous spectacular magazine cover stories about computer software failures during the last decade. Even with these visible lessons in the consequences of poor software, software failures continue to occur on and off the front page. These failures cost the US economy an estimated $59.5 billion per year. [1] An estimated $22.2 billion of the annual losses could be eliminated by software testing appropriately conducted during all the phases of software development. [2]

"Software Testing: Testing Across the Entire Software Development Life Cycle" presents the *first* comprehensive treatment of all *21st Century* testing activities from test planning through test completion for every phase of software under development or software under revision. The authors believe that the cover story business catastrophes can best be prevented by such a comprehensive approach to software testing. Furthermore, the authors believe the regular and consistent practice of such a comprehensive testing approach can raise the industry level of quality that software developers deliver and customers expect. By using a comprehensive testing approach, software testers can turn the negative risk of major business loss into a positive competitive edge.

Many excellent textbooks on the market deeply explore software testing for narrow segments of software development. [3–5] One of the intermediate-level testing textbooks that the authors recommend as a follow-on to this textbook is Dr. James A. Whittaker's *Practical Guide to Testing.* [6] None of these textbooks deal with software testing from the perspective of the entire development life cycle, which

includes planning tests, completing tests, and understanding test results during every phase of software development.

Readers who will benefit the most from this textbook include software professionals, business systems analysts, more advanced Computer Science students, and more advanced Management Information Systems students. The common experience shared by this diverse group of readers is an appreciation of the technology challenges in software development. It is this common experience in software development that will enable the readers to quickly gain a realistic expectation of testing benefits and acknowledge the boundaries of good software testing.

Although this textbook focuses specifically on software testing, fundamental testing concepts presented in the first section apply to all kinds of testing from automobiles to wine. This is possible because, to a large extent, testing is a mindset that anyone can practice on any professional task or pastime.

Computer hardware testers will find about 85% of this textbook directly applicable to their assignments. They should seek additional reference materials for information about the remaining 15% of the techniques they need.

Note: The easiest way to determine whether you are doing software or hardware testing is to examine the recommendation from the test outcome "this system runs too slowly." If the recommendation is to "tweak" the software or buy more/faster hardware, then you are doing software testing. If the recommendation is to reach for the soldering gun, then you are doing hardware testing.

Typically, a person interested in software testing as a profession will begin to specialize in certain kinds of testing like functional testing. Whittaker's textbook mentioned in the beginning of this section can serve as the logical next step for obtaining a deeper understanding of functional testing. The breadth of topics discussed in this textbook should serve as a reminder to the specialists that there are other aspects of testing that often impinge upon the success of their specialty.

1.2 OBJECTIVES AND LIMITS OF TESTING

There are many opportunities for testing in both professional and personal life. We will first explore some examples of non-computer-related testing that show patterns of thinking and behavior useful for software testing. Then we will examine some of the boundaries imposed upon testing by financial considerations, time constraints, and other business limitations.

1.2.1 The Mind of a Tester

Kaner, Bach, and Pettichord describe four different kinds of thinking exhibited by a good tester: [7]

1. Technical thinking: the ability to model technology and understand causes and effects
2. Creative thinking: the ability to generate ideas and see possibilities

3. Critical thinking: the ability to evaluate ideas and make inferences
4. Practical thinking: the ability to put ideas into practice

An example of these kinds of thinking is found in a fable called "The King's Challenge."

The King's Challenge (a fable)

Once upon a time, a mighty king wanted to determine which of his three court wizards was the most powerful.

So he put the three court wizards in the castle dungeon and declared whoever escaped from his respective dungeon cell first was the most powerful wizard in all the kingdom.

(Before reading on, decide what you would do.)

The first wizard immediately started chanting mystical poems to open his cell door.

The second wizard immediately started casting small polished stones and bits of bone on the floor to learn how he might open his cell door.

The third wizard sat down across from his cell door and thought about the situation for a minute. Then he got up, walked over to the cell door and pulled on the door handle. The cell door swung open because it was closed but not locked.

Thus, the third wizard escaped his cell first and became known as the most powerful wizard in all the kingdom.

What kinds of "tester" thinking did the third wizard exercise in solving the king's puzzle?

- Creative thinking: the ability to see the possibility that the door was not locked in the first place
- Practical thinking: the ability to decide to try the simplest solution first

1.2.2 Non-Software Testing at the User Level—Buying a Car

Next, we will use the automobile industry to find non-computer testing examples that can easily be related to software testing. Have you ever shopped for a car or helped someone else shop for a car? What shopping step did you perform first ?

One of the most obvious motivations for testing a car is to determine its quality or functionality before buying one. When you shop for a car, you typically have some pretty specific objectives in mind that relate either to your transportation needs for work or to your transportation needs for recreation. Either way, you are the person who will drive the car, you will be the car "user."

As a user, you are not interested in performing all possible kinds of tests on the car because you assume (correctly or incorrectly) that the manufacturer has done some of those tests for you. The important thing to realize is that you do limit your testing in some way. We will refer to this limited test as a "test drive," although some of the testing does not require driving the car per se. To better understand the *testing limits*, we will first examine what you do *not* test. Then, we will examine what you *do* test before you buy a car.

The following examples of test drive objectives are typically *not* those used for a personal test drive:

Objectives of a Test Drive are NOT

- **to break the car**
- **to improve the car's design**

You do not try to break the car or any of its components. Rather, you seek guarantees and warranties that imply the car manufacturer has already tried to break it and proven the car is "unbreakable" under normal driving conditions for *x* thousand miles or *y* years, whichever occurs first. In other words, you expect the car's reliability to have been already tested by others.

You do not typically try to improve the design of the car because you expect the car manufacturer to have employed a design goal that was reached by the particular model for which you are shopping. If you identify design changes you would like to make in the car, the normal reaction is to simply shop for a different model or for a different manufacturer to find a car with the desired alternative design already implemented.

A software analogy is to shop for a personal accounting package. For example, consider shopping for a home financial tool and finding Quicken by Intuit and Money by MicroSoft. As a user, you are not interested in a "test drive" to break the software. You expect (correctly or incorrectly) that the software is unbreakable. As a user, you are not interested in changing the software design. If you do not like the way Quicken selects accounts using drop-down menus, you consider the way Money selects accounts.

So what *do* you test during a car test drive? Typically, it is determined by your transportation needs (goals). The needs become *test* drive *objectives.* Test objectives are the measurable milestones in testing, which clearly indicate that the testing activities have definitely achieved the desired goals. You translate test drive objectives into testing approaches that validate whether the car on the dealer's lot meets your transportation objectives. Different objectives call for different test drive approaches. Next, we will look at examples of test drive objectives.

Objectives of a Test Drive ARE

- **to validate affordability**
- **to validate attractiveness**
- **to validate comfort**
- **to validate usefulness**
- **to validate performance**

Each of these testing objectives can be validated against the car without trying to break it or redesign it. Some of these testing objectives can be validated even before you get in the car and start the engine.

All of these objectives are personal. You are the only one who can prioritize these objectives. You are the only one who can evaluate the car against these objectives by a test drive, and decide whether to buy the car.

- *Affordability:* down payment, monthly payments, interest rate, and trade-in
- *Attractiveness:* body style, color scheme, body trim, and interior

- *Comfort:* driver or passenger height, weight, body shape, leg room, ingress or egress through a front door or back door, and loading or unloading through a hatchback or rear door.
- *Usefulness:* the number of seats versus the number of passengers, trunk space, convertible hauling space, on-road versus off-road, or trailer hitch weight capacity
- *Performance:* gas mileage, minimum grade of gas required, acceleration for freeway merging, acceleration to beat your neighbor, cornering at low speeds, cornering at high speeds, and the time or mileage between maintenance service

When you have your testing objectives clear in mind, you choose the testing approaches that best validate the car against those objectives. The following examples show some testing approaches and the kinds of testing objectives they can validate.

Testing Approaches Include

- **examining the sticker price and sale contract**
- **trying out the radio, the air conditioner, and the lights**
- **trying acceleration, stopping, and cornering**

These testing approaches are referred to by fairly common terminology in the testing industry.

- Examine = Static testing
  (*observe, read, review without actually driving the car)*
- Try out = Functional and structural testing
  (*work different features of the car without actually driving the car)*
- Try = Performance testing
  (*work different features of the car by actually driving the car)*

1.2.3 Non-Software Testing at the Developer Level—Building a Car

Now, we will switch from the user's, buyer's, or driver's perspective to the auto manufacturer's perspective. As with a shopper, it is important for a car builder to have specific testing objectives in mind and discard other testing objectives that are inappropriate for new car development.

Testing Objectives of a New Car to be Built

- **validate design via scale models.**
- **validate operation of prototypes.**
- **validate mass assembly plans from prototypes.**

The basis for this example is the normal progression of new car development that starts with written *requirements* for a new car such as

- seats six
- carries five suitcases
- runs on regular gas
- consumes gas at a rate of 25 miles per gallon at highway speeds
- has a top speed of 80 miles per hour

These requirements are the nonnegotiable design and manufacturing boundaries set by groups other than the designers such as marketing teams, Federal regulatory agencies, or competitors. It is the auto manufacturer's job to build a new car that does all these things to the letter of the requirements.

With the new car requirements in hand, the test objectives become more understandable. It is the job of the auto design tester to validate the current state of the new car against the car's requirements. If the new car does not initially meet the requirements (as few newly designed cars do), then it is the designer not the tester who must improve the design to meet the requirements.

After design changes are made, it is the tester's job to revalidate the modified design against the requirements. This design, test, correct, and retest cycle continues until the new car design meets the requirements and is completed *before* the car is manufactured.

Hopefully, this discussion points out the advantage of requirements for testing validation at every stage of creating the new car. One of the most pervasive software testing dilemmas today is the decision of companies to build Internet core-business applications for the first time without documenting any requirements. *Note*: Additional requirements testing approaches can be found in the Chapter 6 of this textbook.

As with the user test drive, the manufacture tester has many approaches that can be employed to validate the aspects of a new car against the car's requirements.

Testing Approaches Used While Constructing New Cars

- **plan the tests based on requirements and design specifications.**
- **examine blueprints and clay models.**
- **perform and analyze wind tunnel tests.**
- **perform and analyze safety tests.**
- **perform and validate prototype features.**
- **drive prototype and validate operations.**

This example implies an additional layer of documentation necessary for successful testing. As previously noted, requirements tell the designers what needs to be designed. Specifications (blueprints or models) are the designers' interpretation of requirements as to how the design can be manufactured.

When the specifications are validated against the requirements, all the subsequent physical car assembly validation can be performed against the specifications.

As with the test drive, the car builder testing approaches can be described by common testing terminology.

- Examine = Static testing
  (*observe, read, or review without actually building the car*)
- Perform = Functional and structural testing
  (*work different features of the car models, mock-ups, and manufactured subassemblies*)
- Drive = Performance testing
  (*work different features of the car in the prototypes*)

Because you have probably not built a car, it might be helpful to find examples from a book that details the car-building steps and the manner in which those steps are tested during real car development. [8]

Example of static testing

Read the description of wind tunnel testing that showed changing shapes on the wheel wells would allow the car to achieve 180 mph which became the target speed for road tests later.

Example of test planning

Read the estimate of the number of prototypes to be built for testing the C5, around one hundred, compared with the 300 or more expected to be built for normal car pre-production testing. These prototypes were expected to be used for all static and dynamic (road) testing prior to the start of assembly line production. In fact, some of the prototypes were used to plan and calibrate assembly line production steps.

Read the description of final prototype endurance tests that include driving the test car on a closed track at full throttle for a full 24 hours, stopping only for gas and driver changes.

Examples of functional and structural testing

Read the description of heater and air conditioner testing in which drivers would see how soon the heater made things comfortable in freezing weather. In summer, one internal environment test would let a Corvette sit under the desert sun for 3 hours, then the test driver would get in, close the doors, start the car and air-conditioning to monitor the system until the driver stopped sweating.

Read the description of body surface durability testing which involved driving into a car-wash solution of corrosive salt and chemicals that caused the car to experience the equivalent of a decade of corrosion exposure.

Example of performance testing

Read the description of travel weather testing extremes. Some cars were taken to frigid climates and forced to operate in sub-zero temperatures. Some cars were taken to extremely hot climates and forced to operate in 120+ degree Fahrenheit temperatures.

Read the description of road grade conditions testing that required a driver to pull up a short on a steep slope, set the parking brake, turn off the engine, wait a few moments, then restart the engine and back down the slope.

Read the description of road surface conditions testing where drivers raced over loose gravel to torture the underside of the car and wheel wells.

Read the description of road surface conditions testing that employed long sequences of speed bumps to shake the car and its parts to an extreme.

The book traces all the steps that the General Motors Corvette development team took to create the 1997 model C5 Corvette. It is interesting from the manufacturing standpoint as well as the organizational intrigue standpoint because 1996 was supposed to be the last year the Corvette was made and sold. The C5 became the next-generation Corvette and was brought to market in 1997. The C5 design was manufactured until 2004. Perhaps you have seen the C5 flash by on the highway. It looks like Figure 1.1.

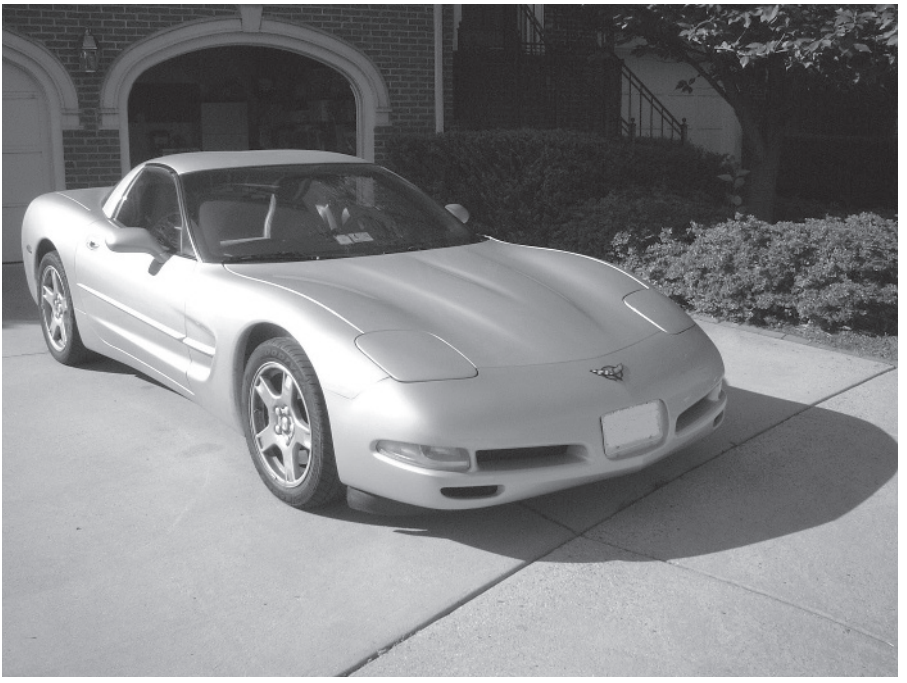

Figure 1.1 1997 Corvette C5 Coupe

1.2.4 The Four Primary Objectives of Testing

Testing can be applied to a wide range of development projects in a large number of industries. In contrast to the diversity of testing opportunities, there is a common underpinning of objectives. The primary motivation for testing all business *development* projects is the same: to reduce the *risk* of unplanned development expense or, worse, the risk of project failure. This development risk can be quantified as some kind of tangible loss such as that of revenue or customers. Some development risks are so large that the company is betting the entire business that the development will be successful. In order to know the size of the risk and the probability of it occurring, a *risk assessment* is performed. This risk assessment is a series of structured "what if" questions that probe the most likely causes of development failure depending on the type of development and the type of business the development must support. This risk motivation is divided into four interrelated testing objectives.

Primary objectives of testing

Testing objective 1: *Identify the magnitude and sources of development risk reducible by testing.*

When a company contemplates a new development project, it prepares a business case that clearly identifies the expected benefits, costs, and risks. If the cost of the project is not recovered within a reasonable time by the benefits or is determined to be a bad return on investment, the project is deemed unprofitable and is not authorized to start. No testing is required, unless the business case is tested. If the benefits outweigh the costs and the project is considered a good return on investment, the benefits are then compared to the risks. It is quite likely that the risks are many times greater than the benefits. An additional consideration is the likelihood that the risk will become a real loss. If the risk is high but the likelihood of the risk occurring is very small, then the company typically determines that the risk is worth the potential benefit of authorizing the project. Again, no testing is required.

If the risk is high and the likelihood of its occurrence is high, the questions "Can this risk be reduced by testing?" and "If the risk can be reduced, how much can testing reduce it?" are asked. If the risk factors are well known, quantifiable, and under the control of the project, it is likely that testing can reduce the probability of the risk occurring. Fully controlled tests can be planned and completed. If, on the other hand, the risk factors are not under control of the project or the risk factors are fuzzy (not well known or merely qualitative), then testing does not have a fair chance to reduce the risk.

Testing objective 2: *Perform testing to reduce identified risks.*

As we will see in subsequent chapters, test planning includes positive testing (looking for things that work as required) and negative testing (looking for things that break). The test planning effort emphasizes the risk areas so that the largest possible percentage of the test schedule and effort (both positive testing and negative testing) are dedicated to reducing that risk. Very seldom does testing completely eliminate a risk because there are always more situations to test than time or resources to complete the tests. One hundred percent testing is currently an unrealistic business expectation.

Testing objective 3: *Know when testing is completed.*

Knowing that 100% testing of the development is unachievable, the tester must apply some kind of prioritization to determine when to stop testing. That determination should start with the positive test items in the test plan. The tester must complete the positive testing that validates all the development requirements. Anything less, and the tester is actually introducing business risk into the development process.

The tester must then complete as much of the risk-targeted testing as possible relative to a cost and benefit break-even point. For example, if there is a $10,000 business risk in some aspect of the development, spending $50,000 to reduce that risk is not a good investment. A rule of thumb is a 10–20% cost to benefit break-even point for testing. If the same $10,000 business risk can be thoroughly tested for $1000–2000, then cost to benefit is very favorable as a testing investment.

Finally, the tester must complete as many of the negative test items in the plan as the testing budget allows after the positive testing and risk testing are completed. Negative testing presents two situations to the test planner:

- The first situation is the complement of the positive test items. For example, if a data field on a screen must accept numeric values from 1 to 999, the values 1, 10, 100, 123, 456, 789, and 999 can be used for positive test completion while the values −1, 0, and 1000 can be used for negative test completion.
- The second situation is the attempt to anticipate novice user actions that are not specified in the requirements or expected during routine business activities. Planning these kinds of tests usually takes deep insight into the business and into the typical ways inexperienced business staff perform routine business activities. The time and expense necessary to test these "outlier" situations often are significantly out of proportion to the likelihood of occurrence or to the magnitude of loss if the problems do occur.

Testing objective 4: *Manage testing as a standard project within the development project.*

All too often, testing is treated as a simple skill that anyone can perform without planning, scheduling, or resources. Because business risk represents real dollar loss, real dollar testing is required to reduce the risk. Real dollar testing means that personnel with testing expertise should be formed into a testing team with access to the management, resources, and schedules necessary to plan and complete the testing. The testing team, as any other business team, can deliver the testing results on time and within budget if the team follows good standard project management practices.

The benefit of this observation is the reassurance that testing does not have to be hit or miss. It can be planned and completed with the confidence of any other professional project to achieve its objectives. The liability of this observation is the realization that testers are a limited resource. When all available testers are scheduled for an imminent testing project, further testing projects cannot be scheduled until you find additional qualified testers.

When you run out of time to test

As with all project schedules, it is possible to run out of testing time. If that situation arises, what can be done to make the most of the testing that you can complete? When

approaching the end of the testing schedule, consider doing a quick prioritization of the outstanding defects. Place most of the testing and correction emphasis on the most severe defects, the ones that present the highest possible business risk. Then review the testing plans that you will not have time to complete and assess the risk that the incomplete testing represents.

Present the development manager with an assessment of the risks that are expected due to the premature halting of testing. The development manager must then decide whether to halt the testing to meet project schedules or to seek additional time and resources to complete the testing as planned.

When you know you can not test it all—positive testing objectives
When you know you can not test it all, review all the completed testing results and compare them with the application or system functionality that the customer has deemed most important. The object of this review is to determine the features to test with your remaining schedule and resources that would make the largest positive impact on the customer's function and feature expectations.

When you know you can not test it all—hidden defect testing objectives
When you know you can not test it all, review the completed testing results and determine if there are trends or clusters of defects that indicate more defects are likely to be found in the same area. Then request a review of that area of code by the development team to determine if additional, hidden defects can be corrected by minor development rework. With minimal additional effort on the part of the developer and tester, likely trouble spots can be addressed before the last remaining testing resources are expended.

1.2.5 Development Axiom—Quality Must Be Built In Because Quality Cannot Be Tested In

Testing can only verify the product or system and its operation against predetermined criteria (requirements). Testing neither adds nor takes away anything. Quality is an issue that is determined during the requirements and design phases by the development project stakeholders or requesting customers. It is not decided at testing time.

1.3 THE VALUE VERSUS COST OF TESTING

Most business decisions are based on a comparison of the value of doing something versus the cost of doing something, typically called the *return on investment* (*ROI*). ROI is the calculation of how quickly and how large the "payoff" will be if a project is financed. If the project will not quickly provide a payoff or the payoff is too small, then the ROI is considered bad. The business motivation for doing something is to receive more benefit than the investment necessary to realize that benefit.

Testing requires the same ROI decision as any other business project. The implication is that testing should be done only when the test results can show benefit

beyond the cost of performing the tests. The following examples demonstrate how businesses have placed value on testing results.

1.3.1 Non-Software Testing at the Marketing Level—Auto Safety versus Sales

Auto manufacturers determined a long time ago that thoroughly testing their new car designs was a safety risk management value that far outweighed the cost of the tests. As a result, the descriptions of new car development safety testing such as those in the Corvette story are found in the literature of all major car manufacturers.

There are two possible outcomes of safety testing and the management of the risk that the tests reveal. The first outcome is the decision whether or not to correct a safety problem before the first newly built car is manufactured and sold in large numbers. The input for this decision is the cost of the safety repair versus the perceived risk of the safety to the public in terms of lawsuits and penalties for the violation of regulations.

The second outcome is the decision whether or not to recall a car already manufactured and sold to many customers in order to fix the safety problem. The inputs for this decision are the presales cost figures and risks and the added cost of retrofitting safety solutions to cars that are already manufactured and sold.

The Ford Pinto is one example of safety risk versus cost to mitigate the risk decision. [9] Ford started selling Pintos in 1971. Later that same year, one of the engineers' testing scenarios discovered that when the Pinto is rear-ended in a collision, the gas tank is punctured which causes an explosion and subsequent fire that can trap occupants in the flaming vehicle. Ford assigned a risk probability to such a rear-end collision and to the subsequent fatalities along with a cost of the risk that would be incurred if families of the fatalities sued Ford.

From the risk assessment, Ford assigned a $25,000 value to a human life lost in a car fire. Then, they estimated the number of car fires that could be expected from the Pintos based on a vast number of car sales statistics. From these two numbers, Ford calculated the break-even settlement cost resulting from faulty gas tank litigation at approximately $2.20 per car. From the manufacturing assessment, Ford calculated the cost of retrofitting every Pinto with a gas tank bracket to be $8.59–11.59 per car. At the end of 1971, Ford decided that the best ROI decision was to refrain from retrofitting the gas tank brackets and pay all faulty gas tank lawsuits.

In nonlife-threatening industries, this risk management strategy might have worked well. In this situation, the families of the fatalities caused by the exploding gas tanks foiled Ford's risk mitigation strategy. Instead of suing Ford individually, the grieving families filed a class action suit after the third such fatality. That forced Ford to reveal its testing discoveries and risk mitigation plan. Instead of the expected $5M–10M in wrongful death lawsuit settlements, an incensed jury hit Ford with a $128M settlement.

1.3.2 Estimating the Cost of Failure

As we saw in the failure example for the Ford Pinto, there are different kinds of business risks and different kinds of business losses that can occur from these risks.

It is important for the tester to understand the different kinds of business losses in order to identify the most appropriate kinds of testing that can mitigate the losses.

Different Kinds of Business Losses

- **revenue or profit**
- **testing resources (skills, tools, and equipment)**
- **customers**
- **litigation**

One of the first measures used by a business to put boundaries around testing is the cost of testing. Regardless of the size of the risk to be reduced by testing, there is a cost associated with performing the tests. Testing does not contribute directly to the bottom-line of a business. Spending $5M on more car testing does not result in an offsetting $5M in increased car sales; therefore, regardless of how well planned and executed the tests are, testing reduces the total profit of the final product.

Unfortunately for project budgets, the cost of testing goes beyond the immediate testing efforts. As the authors of this textbook advocate, good testers need good training, good tools, and good testing environments. These resources are not one-time expenses. Most of these costs are ongoing.

The final two kinds of business losses (customer and litigation) typically represent the highest risk because the cost to the company cannot be forecast as accurately as tester salaries, tools, and facilities. The loss of customers due to perceived issues of poor quality or unsafe products can directly affect the bottom-line of the company, but how many customers will be lost as a result? Part of the answer lies in how the customer developed the negative perception, that is, by trade journal, magazine, newspaper, or TV news commentator, to mention a few ways. To complete the loss cycle, if enough customers develop a negative perception, then large numbers of individual lawsuits or class action suits might result. The loss from litigation might be beyond anyone's ability to imagine, much less to forecast. Finally, at some level the tester must realize that, for test planning purposes, an unhappy customer can do a company as much financial damage as an injured customer.

1.3.3 Basili and Boehm's Rule of Exponentially Increasing Costs to Correct New Software

Managers and executives of companies that develop computer software have perpetuated the myth that quality can be tested into a software product at the end of the development cycle. Quality in this context usually means software that exhibits *zero* defects when used by a customer. It is an expedient myth from a business planning perspective, but it ignores two truths: (1) Testing must be started as early as possible in the software development process to have the greatest positive impact on the quality of the product and (2) You can not test in quality … period!

The reluctance of many managers to include testing early in the development cycle comes from the perception of testing as a "watchdog" or "policeman" ready to pounce on the tiniest product flaw and cause expensive delays in making the

product deadline. Ironically, just the opposite is true. The longer the delay in discovering defects in the software under development, the more expensive it is to correct the defect just prior to software release.

After spending a professional career measuring and analyzing the industry-wide practices of software development, Drs. Basili and Boehm computed some industry average costs of correcting defects in software under development. Figure 1.2 is an irrefutable proof of the axiom "test early and test often." [10A] The numbers first published in 1996 were revalidated in 2001.

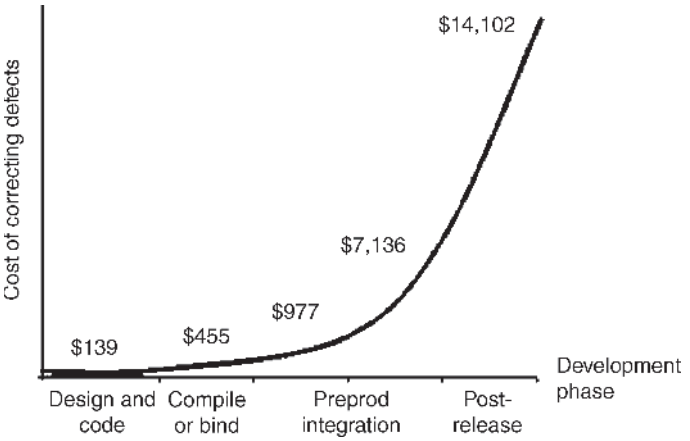

Figure 1.2 Defect correction cost profile for the software industry

At the beginning of a software development project, there is no code, just design documentation. If the design documentation is properly tested (called static testing, see Chapter 6), then the cost of correction is the cost of revising the documentation. This is typically done by a technical writer at relatively small personnel cost after the application-knowledgeable development management provides the correction. Basili finds the average cost to revise a document defect is $25.

As software code (considered fair game for static testing) is written and code execution begins, the cost of a correction rises to $139, primarily due to the expense of programmer effort. Several kinds of testing can be done; however, the code defect resolution at this development phase is correcting the code.

As pieces of program code are completed and tested, they are knitted together, or integrated into larger program modules that begin to perform meaningful business tasks. The cost of correcting these larger units of code more than doubles to $455 due to the additional time it takes to diagnose the problem in more complex code and the additional time needed to disassemble the code module into correctable pieces, correct the code, and then reassemble the code module for retesting.

As the software development team draws closer to the application or product completion, more program modules are brought together into the final delivery package. Capers estimates the cost of defect correction at this stage doubles to $7,136

per defect, primarily due to the increased difficulty in defect diagnosis and correction for this larger aggregation of code that has been packaged for delivery.

Does it really save money to wait and test the software application or product just before it goes out the door? Thirty years of industry statistics say a resounding "NO!"

The story gets worse. If the development manager decides to scrimp on testing or skip testing completely to save a few thousand dollars and "let the customers help test it," the manager will experience the largest project defect correction cost. Capers concludes that it costs on average $14,102 to correct each defect that got past the development team entirely, and that is detected by the customer who receives the application or product. Now the cost of defect correction must also include diagnosis at a distance, package level correction, and the delivery of the correction fixes to *all* customers of the product, not only to the customer who found the defect. If the customer has already installed the application or product and is using it in mission critical situations, then the developer's challenge to fix the customer's code is somewhat like trying to fix a flat tire on a car… while the car is going 50 miles per hour.

1.3.4 The Pot of Gold at the End of the Internet Rainbow

Software that provides businesses with a presence on the Internet can represent billions of dollars in new revenue to a company. This truly staggering business sales increase is possible because the Internet immediately expands the businesses' customer base from a local or regional base to a worldwide base. This phenomenal business sales increase is further possible because the Internet immediately expands the store hours from an 8-hour business day in a single time zone to 24 hours, 7 days per week.

1.3.5 The Achilles Heel of e-Business

The lure of a staggering business sales increase causes business executives to drive a company into its first Internet ventures with a haste born of a king's ransom promise. Everything done by the software development team is scrutinized to find ways to cut corners and save *time-to-market*, to cash in on the king's ransom before the competition does. One of the first software development steps to succumb to the "gold rush" is the proper documentation of requirements. The mandate is, "Just do something … now !!!" Testing takes on the appearance of a speed bump as the executives race toward going live on the Internet.

These business executives either forget or disregard the other side of the equation, that is, the amount of risk in completing such a venture with a new, untried (from the company's perspective) technology. If the company stands to gain billions by the successful completion of the first Internet venture, the company also stands to lose billions if the first Internet venture fails. And failed they did, in droves, during 2000 through 2002.

So, two lessons can be learned from what is known as the "dot.bomb" crash. These lessons can be related surprisingly easily to other instances of companies rushing into new technology markets. First, you can take too many shortcuts when

developing software. Second, you will pay for testing now or later, but the cost of testing is unavoidable. Testing now is always less expensive than testing later.

1.4 RELATIONSHIP OF TESTING TO THE SOFTWARE DEVELOPMENT LIFE CYCLE

Software testing and software development are not totally unrelated activities. The success of both processes is highly interdependent. The purpose of this section is to examine the interdependency of testing and development. Additionally, both testing and development processes are dependent on other support management processes such as requirements management, defect management, change management, and release management. Some of the ancillary management processes that directly impact the effectiveness of testing will be discussed further in Chapter 10.

1.4.1 The Evolution of Software Testing as a Technology Profession

Back in the 1950s and 1960s, software quality was a *hit-or-miss* proposition. There were no formal development processes and no formal testing processes. In fact, the only recorded testing activity during that time was reactive debugging, that is, when a program halted (frequently), the cause was sought out and corrected on the spot.

One of the more famous industry legends of that era was Captain Grace Murray Hopper, the first programmer in the Naval computing center. At that time, computers were composed of vacuum tubes and mechanical switches. One day, Captain Hopper's computer program halted abruptly. After several hours of testing the vacuum tubes and checking the mechanical switches, she found a large moth smashed between two contacts of a relay switch, thereby causing the switch fault that stopped the computer program. She "debugged" the program by removing the moth from the switch. The moth is still on display in the Naval Museum in Washington, DC.

Captain Hopper rose in military rank and professional stature in the software community as she led efforts to standardize software languages and development processes. She was still professionally active and a dynamic speaker in the 1990s.

As more and more software applications were built in the 1960s and 1970s, their longevity enabled many corrections and refinements that yielded very stable, very reliable software. At this juncture, two events occurred that are of interest to testers. First, customers began to expect software to be highly reliable and stable over extended periods of time. Software developers, sensing this growing customer expectation for extremely high-quality software, began to examine the development processes in place and refine them to shorten the incubation time of new software to attain the same stability and reliability as that found in the older, more mature systems.

Software developers of the 1970s and 1980s were, for the most part, successful in capturing their best development practices. These captured practices did provide

a repeatable level of software reliability and stability. Unfortunately for customers, the level of software reliability and stability provided by these captured corporate processes was far below the level of software reliability and stability of the earlier systems. It is informed conjecture that the missing ingredient was a comparable software testing process. For unexplained reasons, this new, lower quality software became acceptable as the industry norm for a large number of computer users. [11]

Testing did not become a recognized formal software process until the 1990s when the Y2K Sword of Damocles threatened all industries that relied on computer power for their livelihood. Testing was thrust to the forefront of frantic software activities as the savior of the 21st century. Billions of dollars were spent mitigating the possible business disasters caused by the shortcuts programmers had taken for years when coding dates. These shortcuts would not allow programs to correctly process dates back and forth across the January 1, 2000 century mark or year 2000 or "Y2K" in the vernacular. The authors think that it is to the credit of the professional testing community that January 1, 2000 came and went with a collective computer whimper of problems compared to what could have happened without intervention. Thousands of businesses remained whole as the calendar century changed. Although some executives mumbled about the cost of all the Y2K testing, wiser executives recognized how close to disaster they really came, and how much of the ability to do business in the 21st century they owed to testers and testing processes.

1.4.2 The Ten Principles of Good Software Testing

Y2K testing did not start in a vacuum. Several groups of computer professionals realized the need to develop a full repertoire of software testing techniques by the mid-1980s. By the 1990s, software testing whitepapers, seminars, and journal articles began to appear. This implies that the groups of the 1980s were able to gain practical experience with their testing techniques.

Although Y2K testing did represent a very specific kind of defect detection and correction, a surprising number of more general testing techniques were appropriate for retesting the remediated (Y2K-corrected) programs. Thus, the Y2K testing frenzy directed a spotlight on the larger issues, processes, and strategies for full development life cycle software testing. These principles are an amalgam of the professional testing experience from the 1980s and 1990s and the Y2K experience to yield the following underlying software testing principles.

Principles of good testing

Testing principle 1: *Business risk can be reduced by finding defects.*

If a good business case has been built for a new software application or product, the majority of the uncontrolled risks can be limited. Indeed, a large part of a good business case is the willingness to chance the risk of failure in a certain market space based on the perceived demand, the competition for the same market, and the timing of the market relative to current financial indicators. With those limits well established, the focus is on the best way and most timely way to capture the target

market. The cost of the needed software development is forecast, usually with some precision, if the effort is similar to prior software development efforts. The question typically missed at this juncture is, "What will it cost if the software does not work as it is advertised?" The unspoken assumption is that the software will work flawlessly this time, even though no prior software development has been flawless. Therefore, a strong connection should be made early in the process between looking for defects and avoiding risk.

Testing principle 2: *Positive and negative testing contribute to risk reduction.*

Positive testing is simply the verification that the new software works as advertised. This seems like common sense, but based on the authors' experience with software during the past 20 years, new off-the-shelf software continues to have defects right out of the package that scream, "Nobody tested me!" There is no reason to expect that new corporate software systems have a better track record. Similarly, negative testing is simply the verification that customers can not break the software under normal business situations. This kind of testing is most often omitted from the software development because it is more time consuming than positive testing; it requires more tester creativity to perform than positive testing, and it is not overtly risk-driven.

Testing principle 3: *Static and execution testing contribute to risk reduction.*

The preponderance of software testing conducted today involves executing the program code under development. Functional, structural (nonfunctional), and performance testing must execute program code to complete the tests. A small but growing number of testing teams and organizations have awakened to the fact that there are a large number of documents produced during software development that, if reviewed for defects (static testing), could significantly reduce the number of execution defects *before* the code is written. The corollary statement is that the best programmers in the organization cannot overcome bad requirements or bad specifications by writing good code.

Testing principle 4: *Automated test tools can contribute to risk reduction.*

As software has become orders of magnitude more complex than the COBOL, PL/1, or FORTRAN systems of yesterday, new types of business risks have arisen. These new risks are most often found in the performance area where system response times and high volumes of throughput are critical to business success. This makes them impossible to test manually. It is true that performance testing tools are quite expensive. It is also true that the potential risk due to poor performance can exceed the cost of the performance test tools by several orders of magnitude. As of 2004, some companies still consider a performance test that involves calling in 200 employees on a Saturday, feeding them pizza, and asking them to pound on a new application all at the same time for several hours. As we will discuss in Chapter 9, this kind of manual testing has severe limitations, including typically an inadequate number of employees that volunteer to test (What happens if you need to test 3000 users and have only 200 employees?) and the nonrepeatability of test results because no one performs a manual test exactly the same way twice. The last 5 years of automated performance test tool maturity has prompted the strong consideration of testing tools to replace other kinds of manual testing when conditions are favorable.

Testing principle 5: *Make the highest risks the first testing priority.*

When faced with limited testing staff, limited testing tools, and limited time to complete the testing (as most testing projects are), it is important to ensure that there are sufficient testing resources to address at least the top business risks. When testing resources cannot cover the top business risks, proceeding with testing anyway will give the system stakeholders the false expectation that the company will not be torpedoed and sunk by software defects.

Testing principle 6: *Make the most frequent business activities* (*the 80/20 rule*) *the second testing priority.*

Once you have the real business killers well within your testing sights, consider the second priority to be the most frequent business activities. It is common industry knowledge that 80% of any daily business activity is provided by 20% of the business system functions, transactions, or workflow. This is known as the *80/20 rule*. So concentrate the testing on the 20% that really drives the business. Because the scarcity of testing resources continues to be a concern, this approach provides the most testing "bang for the buck." The other 80% of the business system typically represents the exception transactions that are invoked only when the most active 20% cannot solve a problem. An exception to this approach is a business activity that occurs very seldom, but its testing importance is way beyond its indication by frequency of use. The classic example of a sleeper business activity is a year-end closing for a financial system.

Testing principle 7: *Statistical analyses of defect arrival patterns and other defect characteristics are a very effective way to forecast testing completion.*

To date, no one has reported the exhaustive testing of every aspect of any reasonably complex business software system. So how does a tester know when the testing is complete? A group of noted statisticians observed a striking parallel between the defect arrival, or discovery patterns in software under development, and a family of statistical models called the Weibull distribution. The good news is that the intelligent use of these statistical models enables the tester to predict within 10%–20% the total number of defects that should be discovered in a software implementation. These models and their ability to predict human behavior (software development) have been around for at least 20 years. The bad news is that we have not found any significantly better ways to develop software during the same 20 years, even though programming languages have gone through multiple new and powerful paradigms. Chapter 12 takes a closer look at these models and how they can assist the tester.

Testing principle 8: *Test the system the way customers will use it.*

This principle seems so intuitive; however, the authors see examples of software every year that simply were not tested from the customer's perspective. The following is a case in point. A major retail chain of toy stores implemented a Web site on the public Internet. Dr. Everett attempted to buy four toys for his grandchildren on this toy store Internet Web site with catalog numbers in hand. Finding the toys to purchase was very difficult and took over 45 min to achieve. When he finally found all four toys and placed them in his shopping cart, Dr. Everett was unable to

complete the purchase. The web page that asked for his delivery address continually responded with the nastygram, "City required, please provide your city name," even though he entered his city name in the appropriate field several different ways, including lowercase, uppercase, and abbreviated formats. In frustration, he abandoned the incomplete purchase. Thinking that he would at least alert the toy store to their Internet problem, Dr. Everett clicked the *Help* button. In the field entitled, "Give us your comments," he described his roadblock to completing the purchase. When he clicked the *Submit* button, a name and address page appeared. Upon completion of the name and address page, he clicked the next *Submit* button, only to receive the "City required, please provide your city name" nastygram again. The application programmers earned an "A" for city field code reuse and an "F" for not testing the city field code in the first place.

An address is a pretty basic piece of customer-supplied business information. The company had a software defect in the customer address code that resulted in the direct loss of business. The defect was such that the company also could not easily learn why they were losing business from their customers. It took this company less than a year to close their Web site because it was unprofitable … perhaps all because nobody tested a city field code routine.

Testing principle 9: *Assume the defects are the result of process and not personality.*

This principle presents an organizational behavior challenge for the tester. Good software developers naturally feel a sense of ownership regarding the programming they produce. Many aspects of the ownership can be positive and can motivate developers to do their best possible work. At least one aspect of the ownership can be negative, causing the developer to deny less-than-perfect results. The tester must find a way to focus on the software defect without seeking to place blame.

Many organizations have started tracking the source of software defects to verify proper matching of programming task with programmer skills. If a mismatch exists, the management process responsible for assigning development teams is truly at fault, not the programmer who is working beyond his or her skill level. If the skills are well matched to the tasks, the question becomes one of providing processes that assist the developer in writing error-free code, that is, programming standards, design walkthroughs, code walkthroughs, and logic-checking software tools. If the execution phase is the first time anyone else on the development team sees the code, the development process provided no safety net for the developer before the code has been executed. In this case, the tester can wear the white hat and, by identifying defects, ultimately assist the improvement of the development process that helps the developers write better code.

Testing principle 10: *Testing for defects is an investment as well as a cost.*

Most executives, directors, and managers tend to view testing only as an expense, and to ask questions such as "How many people? How many weeks delay? How much equipment? and How many tools?" Although these cost factors represent a legitimate part of the overall business picture, so do the tangible benefits that can offset the testing costs, business risk reduction notwithstanding. Some of the benefits can be

realized during the current testing projects by the intelligent use of automated testing tools. In the right situations, automated testing tools can reduce the overall cost of testing when compared with the same testing done manually. Other benefits can be realized on the next testing projects by the reuse of testing scripts and the reuse of defect discovery patterns. When testing scripts are written, validated, and executed, they constitute reusable intelligence for the system being scripted. This "canned" knowledge can be applied to the next version of the same system or to new systems with similar functionality. The technique of reusing test scripts on a subsequent version is called *regression testing*. Defect discovery patterns, when collected over a number of development projects, can be used to more accurately forecast the completion of testing. These same testing histories can also be used to verify that improvements in development processes really do improve the system being developed. Historical defect patterns and their usefulness are explored in Chapter 12.

1.4.3 The Game of "Gossip"

Capers Jones, has some revealing information about the source of software defects. Figure 1.3 shows a plot of his findings on the same software development axis as the defect correction cost plot. [10B]

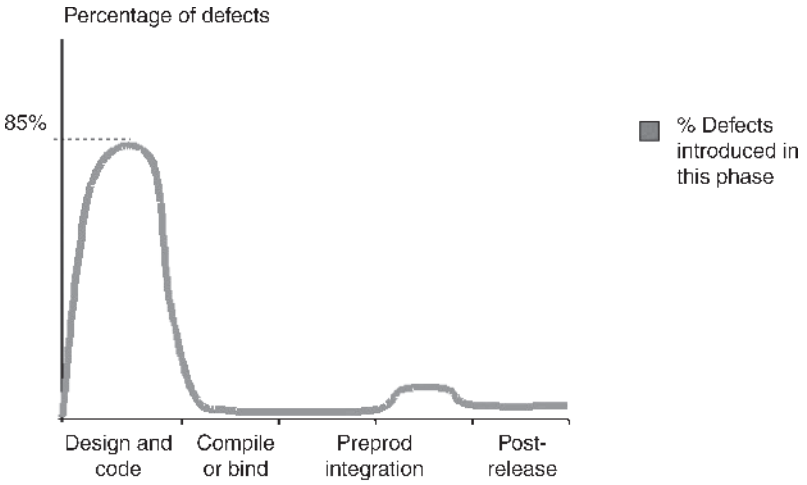

Figure 1.3 Percentage of defects. Applied Software Measurement, Capers Jones, 1996

The findings tell us that 85% of all software defects are introduced at the earliest phase of development before any code has been executed ! If there is no code execution, then what is the source of this mountain of defects? The answer is the documentation, that is, the requirements, the specifications, the data design, the process design, the interface design, the database structure design, the platform design,

and the connectivity design. Intuitively, this seems reasonable. If the system design documentation is incorrect or incomplete, then no programmer can overcome a bad design with good code. A dearth of requirements, the most fundamental development documentation, is endemic to software development. The typical developer's attitude is, "Just tell me what you want and I'll build it."

To demonstrate the fallacy of bypassing requirements documentation, recall the children's game called "Gossip" in which everyone stands around in a circle, and the game leader whispers something to the person on his or her right. That person then whispers the same thing to the person on his or her right, and so on around the ring. When the whispered message makes it around the ring to the last person, the person says the message aloud, and the leader compares it to the original message. Usually, the whole circle of children burst out laughing because the original message was twisted and turned as it went around the circle. Now replace the children in the circle with a business manager who wants a new software system or another product, a director of software development, a software development manager, and say four senior software developers. The director starts the game by whispering his/her new software application requirements to the nearest manager in the circle. Would you bet your company's future on the outcome of the gossip circle? Many companies still do.

1.5 TESTER VERSUS DEVELOPER ROLES IN SOFTWARE TESTING

In the beginning, there was the software developer and he was mighty. He could write specifications, could code programs, could test programs, and could deliver perfect systems. Testers were nontechnical employees who volunteered to come into the office on a weekend and pound on a computer keyboard like a trained monkey in exchange for pizza and beer. The emergence of a massive Y2K catastrophe threat changed technical perceptions forever. The software developer's shiny armor of invincibility was considerably tarnished, whereas the tester's technical acumen rose and shone like the sun. It is now very clear that both the developer and the tester have specific, complementary, highly technical roles to fulfill in the development of good software. This section examines some of the issues around these new roles.

1.5.1 A Brief History of Application Quality Expectations, or "Paradise Lost"

The first software development and operation environments were closed to the end-user. These systems were predominantly batch processing, that is, end-users fed the systems boxes and boxes of daily transactions on punch cards or paper tape and then received the reports the next day or the next week. What happened in between the submissions of batches and the production of reports was considered magic.

If problems occurred on either the input or output side during the batch runs, the end-user never knew it. This closed environment enabled programmers to correct a defect without the end-user's knowledge of the nature of the defect, the nature of the correction, or the amount of time necessary to perform the correction. Therefore, the end-user perceived the system to be perfect. Continued system maintenance over a number of years did, in fact, yield software that was incredibly stable and defect-free.

As the closed system was opened to the end-user via dumb terminals (data display only, no process intelligence like personal computers), the end-user saw how flawlessly this mature software worked. When newer software systems were developed, the systems' immaturity was immediately evident by comparison with the tried and true older systems. Initially, some developers lost their jobs over the poor quality of the new software. End-user pressure to return to the quality of the older systems prompted software development groups to seek and employ development processes for delivering the same software quality. This software was not necessarily better, just consistent in quality. Testing was considered "monkey-work." The authors of this textbook contend that, because testing was held in such low esteem, developers with the best processes soon hit a quality brick wall. The developers' response to end-user complaints of software defects, instability, and unreliability became, "We are using the best development processes in the industry. This is the best we can do."

After a couple of decades of hearing "This is the best we can do," end-users and software customers apparently began to believe it. Still, no professional testing was done. Several books were published about the phenomenon of end-user quality expectations converging downward to meet the software developers' assurance of best effort. Mark Minasi's book, *The Software Conspiracy*, notes the resurging consumer awareness of the relatively poor quality of the new century software. Mark documented a growing consumer constituency that started sending the message, "You *can* do much better" to the software industry through selective product boycotts. [11] Smart software developers began to realize that if they were going to survive in the marketplace, they must team with professional testers to get over the quality brick wall.

To illustrate the point, ask yourself how many times you must reboot your business computer each year. If you reboot more than once or twice a year and have not complained bitterly to your business software retailer, welcome to the world of lower software expectations.

1.5.2 The Role of Testing Professionals in Software Development

Many software professions require very sophisticated technical skills. These professions include software developers, database developers, network developers, and systems administrators. The authors contend that the best software testers must have advanced skills drawn from all of these software professions. No other software

professional except the software architect has a similar need for such a broad range of technical skills at such a deep level of understanding. Without this breadth of technical knowledge and advanced skills, a senior-level software tester could not design, much less execute, the complex testing plans necessary at system completion time for e-business applications.

What does the accomplished software tester do with this broad technical knowledge base? The software tester's singular role is that of a verifier. The tester takes an objective look at the software in progress that is independent of the authors of development documents and of program code and determines through repeated testing whether the software matches its requirements and specifications. The tester is expected to tell the development team which requirements and specifications are met and which requirements and specifications are not met. If the test results are descriptive enough to provide clues to the sources of defects, the tester then adds value to the developer's effort to diagnose these defects; however, the full diagnosis and correction of defects remain solely the developer's responsibility.

What else does a tester do besides validating software? The professional answer is plan, plan, and plan. Testing activities are always short of time, staff, equipment, or all three; therefore, the expert tester must identify the critical areas of software to be tested and the most efficient ways to complete that testing. As with all technical projects, these kinds of decisions must be made and cast into a plan and schedule for testing. Then, the tester must manage the plan and schedule to complete the testing.

1.5.3 The Role of Test Tool Experts in Software Development

Mature automated test tools began to arise in the marketplace around 1995. The good news is that these tools enable software testers to do testing more effectively than by using any manual procedure. In many cases, these tools have enabled software testers to do testing that is impossible to perform manually. Although manual testing still has a place in the software tester's folio of approaches, the use of automated test tools has become the primary strategy.

With over 300 automated test tools in the market, a new testing role emerged that is responsible for identifying the right tool for the right testing, installing the tool, and ensuring that the tool is operating correctly for the test team. The first testing professionals to fill this role tended to specialize in certain kinds of tools from just one or two vendors. As the tool suites grew and matured, the test tool experts found it necessary to broaden their specialty across more tool types and tool vendors. The testing paradigms behind these test tools is examined in Chapter 11.

The impetus behind test tool experts expanding their tool expertise is the software testing community's recognition that no single test tool can support all the different kinds of tests that are necessary across the entire development life cycle.

1.5.4 Who Is on the Test Team?

As with all other software professions, the software testing profession has entry-level skills, intermediate-level skills, and advanced skills. A good test team has a mix of skill levels represented by its members. This enables the more experienced testers to be responsible for the test planning, scheduling, and analysis of test results. The intermediate-level testers can work within the test plan to create the test scenarios, cases, and scripts that follow the plan. Then, with the advice and mentoring of the senior testers, a mix of intermediate-level and entry-level testers executes the tests.

1.6 PUTTING SOFTWARE TESTING IN PERSPECTIVE

Billions of dollars in business are lost annually because companies and software vendors fail to adequately test their software systems and products. These kinds of business losses are expected to continue as long as testing is considered just another checkmark on a "To-do" list or a task given to employees who are on the bench and have nothing else to do.

Testing is, in fact, a professional role that requires technical skills and a mindset that encourages the early discovery of the problems that represent real business risks. Although this textbook covers software testing in detail, many of the testing concepts and techniques it presents can be applied to other engineering disciplines and professions, as well as many personal pursuits.

1.7 SUMMARY

There are many opportunities for testing in both professional and personal life. We first explored some examples of non-computer-related testing that show patterns of thinking and behavior useful for software testing. Then, we examined some of the boundaries imposed upon testing by financial considerations, time considerations, and other business limitations.

1.7.1 The Four Primary Objectives of Testing

Testing can be applied to a wide range of development projects in a large number of industries. In contrast to the diversity of testing scenarios and uses is a common underpinning of objectives. The primary motivation for testing all business development projects is the same: to reduce the risk of unplanned expense or, worse, the risk of failure. This primary motivation is divided into four interrelated testing objectives.

1. **Identify the magnitude and sources of development risk reducible by testing**
2. **Perform testing to reduce identified risk**
3. **Know when testing is completed**
4. **Manage testing as a standard project within the development project**

1.7.2 Development Axiom—Quality Must Be Built In Because Quality Cannot Be Tested In

Testing is concerned with what is in the product or system and what is missing. Testing can only verify the product or system and its operation against predetermined criteria. Testing neither adds nor takes away anything. Quality is an issue that is decided upon during the requirements and design phases by the development project owners or requesting customers. Quality is not decided at testing time.

1.7.3 The Evolution of Software Testing as a Technology Profession

Back in the 1950s and 1960s, software quality was a hit-or-miss proposition. There were no formal development processes and no formal testing processes. Software developers of the 1970s and 1980s were, for the most part, successful in capturing their best development practices. This capture provided a repeatable level of software reliability and stability. Unfortunately for customers, the level of software reliability and stability provided by these repeatable corporate processes was far below the level of software reliability and stability of the earlier systems. It is an informed conjecture that the missing ingredient was a comparable software testing process. For unexplained reasons, this new, lower quality software became acceptable as the norm to a large number of computer users.

Testing did not become a recognized formal software process until the 1990s when the Y2K Sword of Damocles threatened all industries that somehow relied on computer power for their livelihood. Then, testing was thrust to the forefront of software activities as the savior of the 21st century. Billions of dollars were spent mitigating the possible business disasters caused by the shortcuts programmers had taken when coding dates.

1.7.4 The Ten Principles of Good Software Testing

Y2K testing did not start in a vacuum. Several groups of computer professionals realized the need to develop a full repertoire of software testing techniques by the mid-1980s. By the 1990s, software testing whitepapers, seminars, and journal articles began to appear. This indicates that the groups of the 1980s were able to gain practical experience with their testing techniques.

Although Y2K testing did represent a very specific kind of defect detection and correction, a surprising number of more general testing techniques were appropriate for retesting the remediated (Y2K-corrected) programs. Thus, the Y2K testing frenzy directed a spotlight on the larger issues, processes, and strategies for full development life cycle software testing. These principles are an amalgam of the professional testing experience from the 1980s and 1990s and the Y2K experience to yield the following underlying software testing principles.

Principles of good testing

1. **Business risk can be reduced by finding defects.**
2. **Positive and negative testing contribute to risk reduction.**
3. **Static and execution testing contribute to risk reduction.**
4. **Automated test tools can substantially contribute to risk reduction.**
5. **Make the highest risks the first testing priority.**
6. **Make the most frequent business activities (the 80/20 rule) the second testing priority.**
7. **Statistical analyses of defect arrival patterns and other defect characteristics are a very effective way to forecast testing completion.**
8. **Test the system the way customers will use it.**
9. **Assume that defects are the result of process and not personality.**
10. **Testing for defects is an investment as well as a cost.**

KEY TERMS

Test limits
Testing objectives
Multiple testing approaches
Testing completion
Verifier
Requirements
Development risk
Risk assessment
Return on investment (ROI)
80/20 rule

KEY CONCEPTS

- Testing is a technical profession with a significantly different mindset, and with significantly different concepts and skills from those of the technical developer profession.
- All engineering projects introduce defects into the new system or product. Making a business decision not to find the defects, by not testing, will not make them go away.

- Testing should start at the beginning of a development project because everything produced by the project is an excellent test candidate.
- To be most effective, testing plans and activities should focus on known business risk.
- No business testing project can ever exhaustively test for every possible defect in a product or service because of finite limitations in funds, skills, and resources.
- The real challenge for experienced testers is to identify what can go untested with the least impact to the business.