> The awareness of the ambiguity of one's highest
> achievements (as well as one's deepest failures)
> is a definite symptom of maturity.
>
> **—Paul Tillich (1886–1965)**

Chapter 1

Failure

Fail better.

—Samuel Beckett,
"Worstward Ho," 1983

Civilizations perish, organisms die, planes crash, buildings crumble, bridges collapse, peaceful nuclear reactors explode, and cars stall. That is, systems fail. The reasons for failure and the impacts of the failure vary; but when systems fail, they bring forth images of incompetence. Should we expect perfection from systems we develop when we ourselves are not perfect?

All man-made systems are based on theories, laws, and hypotheses. Even core physical and biological laws contain implicit assumptions that reflect a current, often limited, understanding of the complex adaptive ecosystems in which these systems operate. Surely this affects the systems we develop.

Most people recognize the inevitability of some sort of failure although their reaction to it is based on the timing of the failure with respect to the life expectancy of the system. A 200-year-old monument that crumbles will not cause as much consternation as the collapse of a two-year-old bridge.

In software, failure has become an accepted phenomenon. Terms such as "the blue-screen-of-death" have evolved in common computer parlance. It is a sad reflection on the maturity of our industry that system failure studies, causes and investigations, even when they occur, are rarely shared and discussed within the profession. There exist copious amounts of readily available literature analyzing failures in other industries such as transportation, construction, and medicine. This lack of availability of information about actual failures prevents the introduction of improvements as engineers continue to learn the hard way by churning out fail-prone software. Before arguing that a software system is more complex than most mechanical or structural systems that we build, and that some leeway should be given, there is a need to look at the various aspects of failure in software.

A Formal Definition of Failure

Although the etymology of "failure" is not entirely clear, it likely derives from the Latin *fallere*, meaning "to deceive." Webster's dictionary defines failure as "omission of occurrence or performance; or a state of inability to perform a normal function." It is important to note that it is not necessary that the entire system be non-operational for it to be considered a failure. Conversely, a single "critical" component failing to perform may result in the failure of the entire system. For example, one would not consider a car headlamp malfunction a failure of the car; however, one would consider a non-working fuel injection system as a failure. Why the distinction? Because the primary job of a car is conveyance. If it cannot do that, it has failed in its job. The distinction is not always so clear. In some cases, for example, while driving at night, a broken headlamp might be considered a failure. So how do we define software failures? Is a bug a failure? When does a bug become a failure? In software, failure is used to imply that:

- *The software has become inoperable.* This can happen because of a *critical* bug that causes the application to terminate abnormally. This type of failure should never happen in an end-customer scenario. Most organizations have mandatory guidelines about QA (quality assurance) — that no software should ship to a customer until all *critical* and *serious* bugs have been fixed (see Chapter 12 on quality). All things being equal, such a failure usually signifies that there is a problem outside the software boundary — the hardware, external software interacting with your software, or some viruses.
- *The software is operable but is not delivering to its desired specifications.* This may be due to a genuine oversight on the part of the development team, or a misunderstanding in the user requirements. Sometimes these requirements may be undocumented, such

as user response times, screen navigations, or the number of mouse-clicks needed to fulfill a task. By our definition, a critical requirement *must* be skipped in development for the software to be considered a failure. Such a cause of failure points to a basic mismatch of the perceived criticality of needs between the customer and the project team.

- *The software was built to specifications but has become unreliable to a point that its use is being impacted.* Such failures are typical when software is first released, or a new build hits the QA team (or the customer). There could be many small bugs that hinder the user from performing his or her task. Often, developers and the QA team tend to overlook inconsistencies in the GUI (graphical user interface) or in the presentation of data on screens. Instances of these inconsistencies include, but are not limited to the placement of "save" and "cancel" buttons, format of error messages, representation of entities across screens (e.g., "first name" followed by "last name" in one screen, and "last name, first name" in another). These cannot be considered bugs; rather, they are issues with a nuisance value in the minds of users, "putting them off" to a point that they no longer consider the software viable.

Failure Patterns

Failure as a field has been studied extensively ever since machines and mechanical devices were created. In the early days, most things were fixed when they were broken. It was evident that things would fail at some point in their lifetime, and there had to be people standing by who could repair them. In the 1930s, more studies were conducted, and many eminent scientists (including Waloddi Weibull) proposed probabilistic distribution curves for failure rates. The most commonly accepted is the "bathtub curve," so called because its shape resembles a Victorian bathtub (Figure 1.1).

The first section of the curve is called the "infant mortality." It covers the period just after the delivery of a new product and includes early bugs and defects. It is characterized by an increasing and then a declining failure rate. The second section, the flatbed of the bathtub, covers the usual failures that happen randomly in any product or its constituents. Its rate is almost constant. The third section is called "wear-out," in which the failure rate increases as the product reaches the end of its useful life. In physical terms, parts of a mechanistic system start to fail with wear and tear.

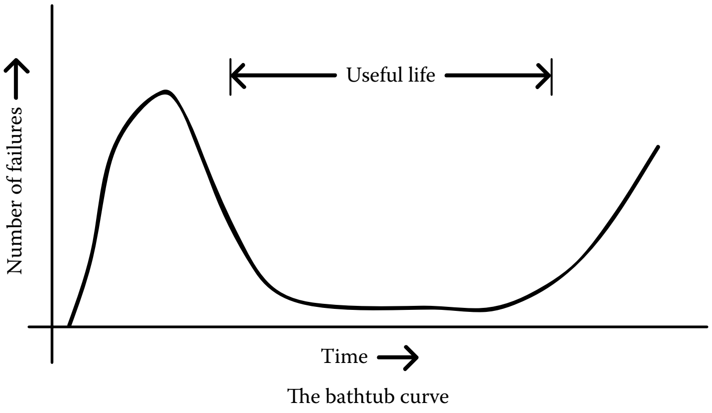


Figure 1.1 The bathtub curve.

Until the 1970s, the bathtub curve remained the basis of scheduled maintenance for physical machines and production plants, where parts of such systems were regularly replaced regardless of their wear and tear. All this changed in 1978 due to the work of Nowlan and Heap. They worked with many years of data obtained from the airline industry and disproved the concept of a *defined useful life in systems*. Their studies revealed (Figure 1.2) that greater than 70 to 90 percent of all failures were random, leaving only a very small percentage of age-related failures. These failure patterns made it clear that scheduled maintenance could have the reverse effect on the life of a system. Any intrusion (part replacement) could potentially cause more damage because the system would reset to its infant mortality failure rate (which is higher) from its random failure rate. Nowlan and Heap created the concept of Reliability Centered Maintenance, a concept later extended to other systems by Moubray in 1997. His idea is that as more bugs and problems accumulate in a system, its reliability starts deteriorating. As experienced managers and engineers, we need to know when to schedule component replacements and perform system maintenance; and Chapter 12 (on quality) further discusses this.

The Dependability of a Software System

Dependability is a combination of reliability — the probability that a system operates through a given operation specification — and availability — the probability that the system will be available at any required instant. Most software adopts the findings of Nowlan and Heap with respect to failures. During the initial stages of the system, when it is being used

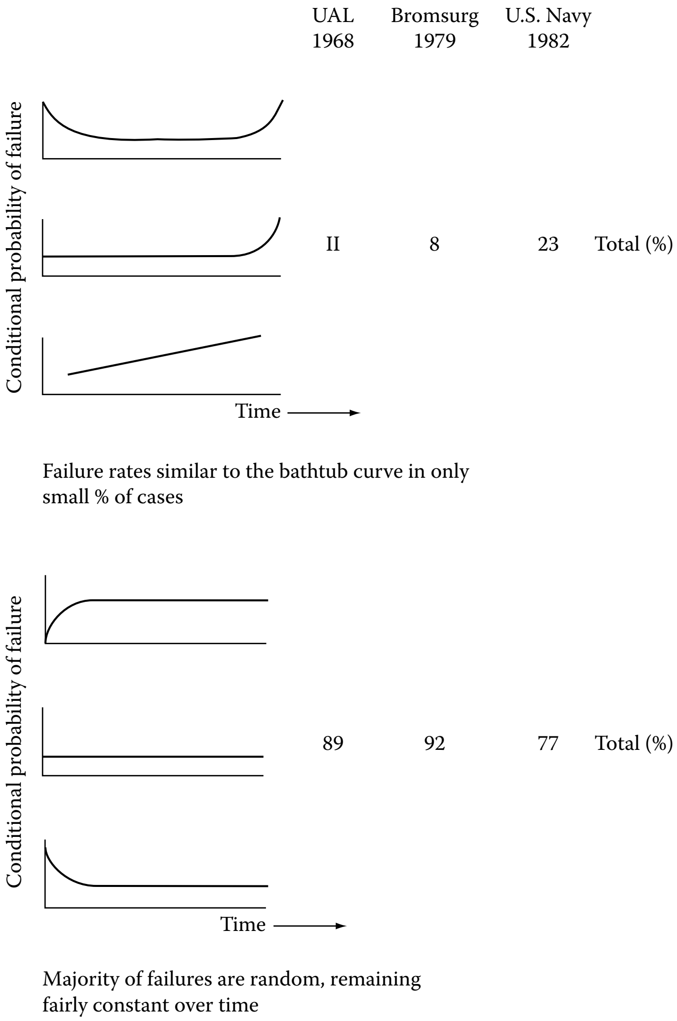


Figure 1.2 Nowlan and Heap's findings.

fresh in production, the failure rates are high. "Burn-in" is a concept used in the electrical and mechanical industries, wherein machines run for short periods of time after assembly so that the obvious defects can be captured before being shipped to a customer. In the software industry, alpha and beta programs are the burn-in. In fact, most customers are wary of accepting the first commercial release (v 1.0) of a software product. The longer a system is used, the higher the probability that more problems have been unearthed and removed.

For most of its useful life, the usage of any software in the production environment is very repetitive in nature. Further, because there is no physical wear and tear, the rate at which problems occur does remain the same. The reality, however, is that during the course of its usage, there is a likelihood that the software system (and the ecosystem around it) has been tinkered with, through customizations, higher data volumes, newer interfaces to external systems, etc. Each time such changes occur, there is the chance that new points of failure will be introduced. As keen practitioners, one should be wary of such evolving systems. Make a call regarding the migration of the production environment (see Chapter 11 on migration). This call will be based on more abstract conditions than simply the rate at which bugs are showing up or their probability — it will depend on one's experience in handling such systems.

Known but Not Followed: Preventing Failure

Software systems fail for many reasons, from the following of insufficient R&D processes during development to external entities introducing bugs through a hardware or software interface. Volumes have been written about the importance of and the critical need for:

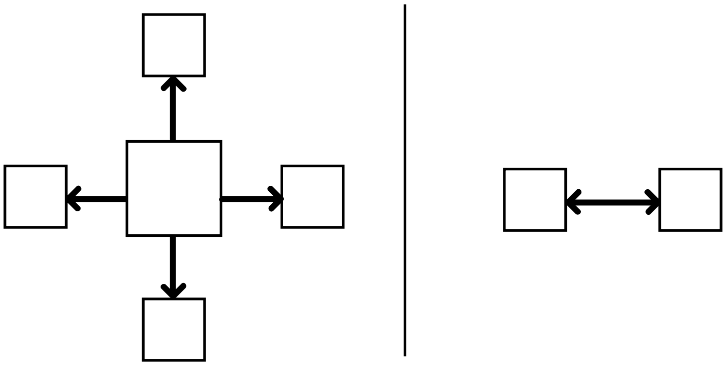
- A strict QA policy for all software development, customization, and implementation
- Peer and buddy reviews of specifications, software code, and QA test plans to uncover any missed understanding and resource-specific flaw patterns
- Adhering to documentation standards and following the documentation absolutely
- Detailed end-user training about the system, usage, maintenance, and administration
- Feedback from user group members who will interface with the system and use it on a day-to-day basis
- Proper availability of hardware and software infrastructure for the application based on the criticality of its need
- Frequent reviews, checks, and balances in place to gather early warning indicators about any untoward system behavior

There is little need to reiterate the importance of this list. The question one wants to tackle is: if these things are so clearly recognized, why do we continue to see increasing incidence of failure in software systems? Or do these policies come with inherent limitations because of which they cannot be followed in most software life cycles?

Engineering Failures: Oversimplification

Dörner, in his book entitled *The Logic of Failure*, attributes systems failure to the flawed human thinking process. Humans are inherently slow in processing various kinds of information. For example, while one can walk and climb stairs with ease (while scientists still struggle to remove the clumsiness in the walking styles of smart, artificial intelligence-based robots), one is not as swift in computing primitive mathematical operations such as the multiplication of two large real numbers. It is this apparent slowness that forces one to reduce analytical work by selecting one central or a few core objects. We then simplify, or often oversimplify, complex interrelationships among them in a system. This simplification gives one a sense that things are in one's control and that one is sufficiently competent to handle them (Figure 1.3). On one hand, this illusion is critical for humans to maintain their professional self-esteem. On the other hand, it forces one to make choices that may not be right for the overall system. And because one is unable to think too far ahead, and unwilling to comprehend very complex temporal behaviors of the system one is proposing to create at the onset, one inevitably leaves pitfalls in the system that can cause its failure.

It is a good idea to regularly document the assumptions made at various stages of design and development, and to have them reviewed by domain experts or actual users. A building's structural engineer not only needs specifications about the layout of the proposed building, but also of the



Human beings often oversimplify complex relationships by selecting one central object or a few core objects because it reduces analytical work

Figure 1.3 Simplifying relationships.

land orientation, soil composition, water table, wind (or snow) or environmental conditions, etc. So while the building code for earthquakes may not require anything exceptional if the land is far from a fault line, the presence of an active railway track nearby may result in similar conditions and thus must be considered *a priori*. Mission-critical software has been known to fail because of trivial assumptions such as measurement units (e.g., meter-kg-second instead of foot-pound-second). Developers may have found it difficult to create a system with definable units, or perhaps they were unaware of measurement systems used in different parts of the world. Assumptions that are wrong are a bane of the software world.

Execution Failures: Overruns under Pressure

Often, software does not even reach a stage where it is put in front of users; the project fails before the software is released. The causes of such failures are usually time, resource, or scope overruns. Many times, project managers and architects are not 100-percent aware of the domain's complexity or the new technologies that would be used for the proposed system. Such failures are common in porting projects (see Chapter 11 on migration) where managers tend to estimate project costs on the basis of the origin and destination technologies without looking into the details of the features used. The complexity involved in porting a system from Microsoft SQL Server DB to Oracle DB can vary immensely if triggers, stored procedures, and DB-dependent system calls have been used extensively.

Functional Failures

Unclear user requirements and undocumented "understandings" between the client and the sales team of the vendor are causes of project cost overruns. In their enthusiasm to close a deal, meet and beat their sales quota, or because sales people do not have an adequate understanding of what it will take to deliver the software, project costs may be severely under-quoted. It is advisable to have a technical person involved with the sales process, before the final estimates are shared with the client. All expectations should be documented as formal features in the requirement specifications. Sometimes the client innocently adds requirements, often immediately before signing the deal. These can cause several problems. Nontechnical verbiage such as "one of my sales staff works from another city and will need a copy of the database on his computer" or "we have some data in our existing small computer we used, which you can move to your new system" can completely alter delivery times.

Failing to follow a pre-decided formal change management process is another cause for delay. A potential problem occurs in the development of a system when a programmer needs to implement an obscure or vague specification. Consultants and experts in the domain may not be accessible in time to get advice or clarification. If clarification is not sought or received, any guesswork on the part of the programmer can lead to unwanted results, and even cause system failure. For example, the programmer may have specifications that fields, which are not explicitly filled by the user on a screen, should default to certain values when the user hits the "save" button. When a change is introduced, adding a new field to the screen, the programmer may not get a clear specification about its default value and decides to select the one "that makes sense." This could lead to problems in the data being displayed elsewhere in the application.

Ecosystem Failures: Uncoordinated Efforts within a Team

Improper or uncoordinated marketing and sales processes also can lead to the failure of a product (Figure 1.4). Take, for example, the failure of online grocery and furniture stores the first time they were launched in early 2000. Although 40 percent of their budget was spent in marketing, they missed out on two critical customer shopping behaviors: (1) customers like to touch and feel before they buy; and (2) walking into a store and buying groceries or furniture is not so negative an experience that people would flock to an online store. In fact, in the case of furniture, it was sometimes a family outing that stimulated the purchase. Similar examples exist for huge marketing blunders such as the introduction of Coke in China (in Mandarin, Coke translated as "female horse stuffed with wax"), and the unsuccessful launch of Nova by General Motors in Mexico (which meant "goes nowhere"). These companies spent millions before realizing their mistakes.

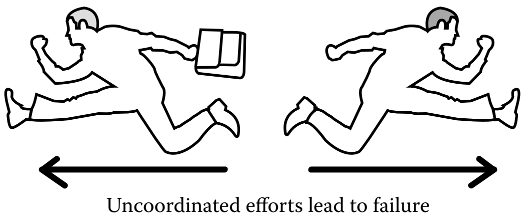


Figure 1.4 Lack of coordination.

Although projects are executed flawlessly, with good sales and marketing strategies, and there is customer demand for the product, companies may forget to plan for the right logistical and delivery systems. Online grocery stores went out of business because they did not sort out their delivery systems. For example, they had a difficult time managing deliveries on congested roads, to a widespread customer base. Their costs never came close to their theoretical efficiencies, and they were not able to maintain promised delivery schedules. Other E-commerce companies became successful only when they collaborated with local delivery establishments. Internet shoppers dropped in number after receiving delayed or poor service from online retailers at the turn of the century. Cases of customers getting a battery delivered to them on Christmas and the toy a week later were commonplace.

Natural disasters (e.g., floods, earthquakes, fires) and man-made problems (e.g., terrorism, war) can cause logistical issues and should be taken into account in making disaster recovery plans for critical applications. Most customers understand and are realistic in their needs during such abnormal times, as long as they see the company has things under control and is working to provide them with service free of interruptions.

Changing Playing Fields Cause Unprecedented Failure

Sometimes, parameters external to a business can have a detrimental impact on its products. As a software architect, one should always have an eye on the changing technology landscape (refer to McLuhan in Chapter 2 on systems). Technology changes can turn an offering obsolete overnight. When cell phones overran pagers, it took Motorola — the leader in pagers— a long time to catch up with Nokia in the cell phone market.

Other things, such as economic sanctions between countries, due to politics, can affect the success of software products, especially if they have an export-oriented market. All one can do is hope that political leaders maintain amicable relationships with the rest of the world.

Faulty Leadership Causes Failures

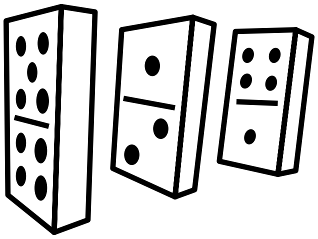
This facet may be out of one's control but leadership is a good indicator of the future success of a software project. Fickle-minded leaders, lacking the vision or the mettle to drive their ideas, hold back an otherwise successful company. Good leaders know when to stand firm and when to give in. Often, project managers are not strong enough to say "no" to a customer. Acceptance of changes late in the software development life cycle leads to scope creep, cost overruns, long working hours, and a generally unhappy, unmotivated team.

Sometimes there is management pressure to rush through the release of a product. In such cases, it is advisable to guide management with regard to any compromises that one will have to make that may affect the safety of the system. Management can then make a collective decision on the software release dates. It is better to deal with bad information as early as possible. Keeping quiet under pressure will only complicate the problem. In fact, it is quite likely that the cost will multiply if the problem has to be resolved at a customer site rather than in the development lab.

Techno-political factors rear their heads more often when external consultants and contractors are involved. Because their involvement in the project is for the duration of the contract, there could be a tendency on the part of less than professional contractors to ignore long-term consequences of failure, as the contractors would have moved on by then. The project manager should create an environment in which each contributor is committed to success, irrespective of the length of his or her attachment to the project.

Cascading Failures

As the name implies, cascading failures (Figure 1.5) begin with the failure of a small component, which in turn triggers the failure of other components, leading sometimes to the failure of the entire system. Quite often, such failures are the result of faulty fail-over implementation strategies. With fail-over, there is an effort to redistribute the load from a failed node, which results in an excess load on other components beyond their capacity. This may cause them to fail and may further aggravate the situation. The effect snowballs and soon the entire network may also fail.



Failures can cascade

Figure 1.5 Cascading failures.

Detection of such failures, at their very onset, is the key to preventing cascading failures. There are complex algorithms in distributed computing under Byzantine failure and Byzantine fault tolerance.

Monitoring Systems

Systems are created to deal with any number of things. Sometimes they deal with extremely dangerous situations, for example, nuclear reactors and space shuttles. It is very difficult to test these systems for failures because the failures in either case would have catastrophic impacts. Thus, these systems must be run through hypothetical failure scenarios and recovery mechanisms. Essential components in these systems include monitoring systems for the detection and reporting of failures, and emergency control functions that will make intelligent decisions by switching control to safe zones when faults are detected. In some cases this may even include human intervention.

Software systems should learn from this. Routine checks of the system should be mandatory. Browsing system logs periodically, even when users have reported no critical or serious failures, is a good exercise. It is also helpful to have monitoring software built into all server components to automatically check the health of the component periodically. It is important to remember that detecting failures, on a few server components, can prevent the spread of those failures to the entire system. Some techniques used in networking include checksums, parity bits, software interlocks, watchdog timers, and sample calculations. Sample calculations are beneficial when writing code for some critical function that may or may not require mathematical operations or multiprocessor systems. It involves doing the same calculation twice, at different points in time on the same processor or even building software redundancy by writing multiple versions of the same algorithm being executed simultaneously and verified for identical results.

Reliability in Software

Dimitri Kececioglu introduces a formal definition for this:

> "Reliability engineering provides the theoretical and practical tools whereby the probability and capability of parts, components, equipment, products and systems to perform their required functions for desired periods of time without failure,

> in specified environments and with a desired confidence, can be specified, designed in, predicted, tested and demonstrated."

In material product engineering, reliability is extremely important. Manufacturers want to reduce their costs as much as possible, sometimes taking the route of using inferior quality raw materials, or not adhering to approved production standards. This may not be the right strategy because it can increase the cost of supporting the product once it gets out in the field.

Although the above may have been used in the context of purely mechanical systems, it also has relevance to the software industry. Conscious efforts must be made to weave reliability through the life cycle of the development by incorporating reliability models in one's software right from the product planning stage. The architecture and design (scalability, performance, code reuse, third-party components, etc.) set the stage for the reliability of the product. Further along, coding styles, adherence to good software engineering practices, and adequate developer testing help in creating reliable code. Subsequent testing, using meaningful in-house data and field data (if the customer is using a system already), helps in ascertaining how reliably the software will work in a production environment.

An analysis of bugs discovered in the software, which requires proper documentation of all bugs found in-house or at the customer site, irrespective of their severity (see Chapter 12 on quality), is another valuable tool. Reliability engineering is known to use various distribution models — Weibull being the most widely used — to get a good handle on the success rates of manufactured products, and ensure that they balance the business goals of the customer.

Accountability for Failure

Systems fail due to human error. Typical errors are often the result of incomplete information to handle the system, or the failure to follow a long set of procedures while using a system. Failures can also occur when accountability has not been properly established in the operational hierarchy of the system. If users or operators are not fully aware of their roles and responsibilities (especially those who are monitoring the behavior of the system), they tend to assume that reporting a potential problem may not be *their* job. They assume that the other guy or the manager will ultimately spot it. Critical problems can easily be missed this way. The following is a good checklist to ensure that no part of the system is passed over:

- Consider all critical components in a system.
- If a component has more than one mode of operation, each of these should be considered individually.
- Estimate the probability of failure based on:
  - Historical data for similar components in similar conditions
  - QA bugs analysis
  - Experience of one's earlier projects
  - Documentation by industry experts in their usage scenarios
- Remember that failures in a particular operation can take a number of forms.
- Assign good people (refer to Chapter 8 on process) to these critical components — during development, QA, and their eventual operation.

Liability of Failure

Pilots will refuse to fly an aircraft if they have the slightest suspicion of a technical snag, and a builder will always be held liable for not following construction norms. Who is liable for the failure in a software system? If the customer does not have a maintenance contract, chances are that it will be expensive to get the services of the software company. It is this lack of liability that has contributed to the lack of serious analysis of failures in software systems.

Getting out of Failures

Two approaches are used: (1) fail-over and (2) fail-safe (Figure 1.6). Failover means detecting a system failure and migrating the functionality of the failed system to a backup system. For example, in most infrastructure networks (e.g., power grids or telecom lines), the loads carried by each substation or node are dynamically redistributed. The strategy states that that if a node is unavailable due to failure, the load it carries is rapidly distributed to the other nodes on the network. Similarly, in fault-tolerant machines or redundant servers, there is more than one intelligent controller connected to the same device. In the event of a controller failure, failover occurs and the survivor takes over its I/O load.

The fail-safe approach, on the other hand, deals with prevention. Such systems have the in-built capacity that any component failure triggers an action, or set of actions, that will always bring the system to a safe situation. Sometimes this is achieved through a process that uses controls and logic to prevent an improper operation or transaction from occurring and produces an error condition or defect instead of failure. Checks for

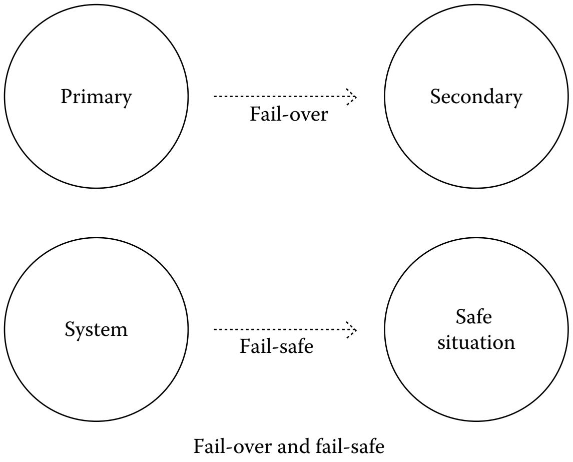


Figure 1.6 Avoiding failures.

boundary conditions (array bounds, divide by zero) are examples of such fail-safe mechanisms.

Product and Project Failures

Product development involves one or more projects, each of which has its own life cycle. A failure of the latter — that is, project failure — could be the result of inaccurate initiation, choosing a wrong life cycle, inadequate resourcing or funding, contractual disagreements, etc. An important observation to remember is that products may fail due to inadequacies in the projects driving them. Interestingly, these inadequacies do not cause project failures themselves. More details on product and project failures are discussed in Chapter 7 on life cycles.

Summary

Failure as a field has been studied extensively ever since machines and mechanical devices were created. There are many degrees of failure: the software could become inoperable; it could be operable but not deliver to its desired specifications; it may have been built to specifications but become unreliable to a point that its use is being impacted. Some of the reasons why software applications fail include insufficient R&D (research and development) processes during development and the natural tendency of humans to oversimplify complex interrelationships among the core entities in a system. Just as products can fail, so can projects. Project failure is often due to inadequate change control management. Fortunately, failure has its patterns. The probability of failure can be reduced through better design and engineering. As a designer, one should incorporate reliability models in one's software. As for handling failures, one can take the fail-over and fail-safe approaches.