Chapter 2

Testing Basics

1. CONTEXT OF TESTING IN PRODUCING SOFTWARE

Almost everything we use today has an element of software in it. In the early days of evolution of software, the users of software formed a small number compared to the total strength of an organization. Today, in a typical work place (and at home), just about everyone uses a computer and software. Administrative staffs use office productivity software (replacing the typewriters of yesteryears). Accountants and finance people use spreadsheets and other financial packages to help them do much faster what they used to do with calculators (or even manually).

- **Everyone in an organization and at home uses e-mail and the Internet for entertainment, education, communication, interaction, and for getting any information they want.**
- **In addition, of course, the "technical" people use programming languages, modeling tools, simulation tools, and database management systems for tasks that they were mostly executing manually a few years earlier.**

The above examples are just some instances where the use of software is "obvious" to the users. However, software is more ubiquitous and pervasive than seen in these examples. Software today is as common as electricity was in the early part of the last century. Almost every gadget and device we have at home and at work is embedded with a significant amount of software.

- **Mobile phones, televisions, wrist watches, and refrigerators or any kitchen equipment all have embedded software.**

Another interesting dimension is that software is being used now in mission critical situations where failure is simply unacceptable. There is no way one can suggest a solution of "please shutdown and reboot the system" for a software that is in someone's pacemaker! Almost every service we have taken for granted has software.

- **Hanks, air traffic controls, cars are all powered by software that simply cannot afford to fail. These systems have to run reliably, predictably, all the time, every time.**

This pervasiveness, ubiquity, and mission criticality places certain demands on the way the software are developed and deployed.

- **First,** an organization that develops any form of software product or service must put in every effort to drastically For instance, imagine finding a defect in the software embedded in a television after it is shipped to thousands of customers. **How is it possible to send "patches" to these customers and ask them to "install the patch?"** Thus, the only solution is to do it right the first time, before sending a product to the customer.
- **Second,** defects are unlikely to **remain latent for long**. When the number of users was limited and the way they used the product was also predictable (and highly restricted), it was quite possible that there could be defects in the software product that would never get detected or uncovered for a very long time. However, with the number of users increasing, the chances of a defect going undetected are becoming increasingly slim. If a defect is present in the product, someone will hit upon it sooner than later.
- **Third,** the nature of usage of a product or a service is becoming increasingly unpredictable. **When bespoke software is developed for a specific function for a specific organization (for example, a payroll package),** the nature of usage of the product can be predictable. For example, users can only exercise the specific functionality provided in the bespoke software. In addition, the developers of the software know the users, their business functions, and the user operations. On the other hand, consider a generic application hosted on the Internet. The developers of the application have no control over how someone will use the application. They may exercise untested functionality; they may have improper hardware or software environments; or they may not be fully trained on the application and thus simply use the product in an incorrect or unintended manner. Despite all this "mishandling," the product should work correctly.
- **Finally,** the consequence and impact of every single defect needs analysis, especially for mission critical applications. It may be acceptable to say that **99.9% of defects are fixed in a product for a release, and only 0.1% defects are outstanding.** It appears to be an excellent statistics to go ahead and release the product. However, if we map the 0.1% failure in mission critical applications, the data will look like this.

  - ✓ **A total of 10,000 incorrect surgery operations per week.**
  - ✓ **Three airplane crashes every day.**
  - ✓ **No electricity for five hours every week.**

For sure, the above data is unacceptable for any individual, organization, or government. Providing a work around, such as "In case of fire, wear this dress," or documenting a failure, such as "You may lose only body parts in case of a wrong airplane landing" would not be acceptable in cases of mission critical applications. These interactions and their impact are captured in the grid in **Figure 1.1.**

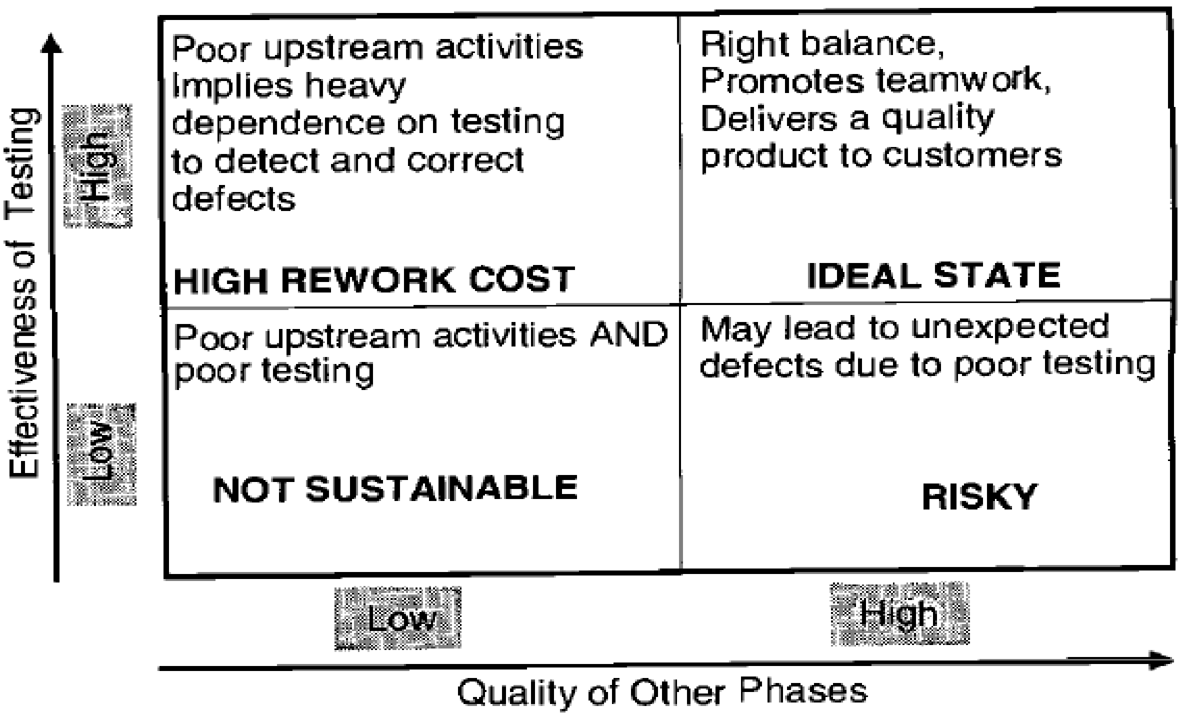

Poor upstream activities Implies heavy dependence on testing to detect and correct defects
HIGH REWORK COST
Right balance, Promotes teamwork, Delivers a quality product to customers
IDEAL STATE
Poor upstream activities AND poor testing
NOT SUSTAINABLE
May lead to unexpected defects due to poor testing
RISKY
Effectiveness of Testing
High
Low
Low
High
Quality of Other Phases

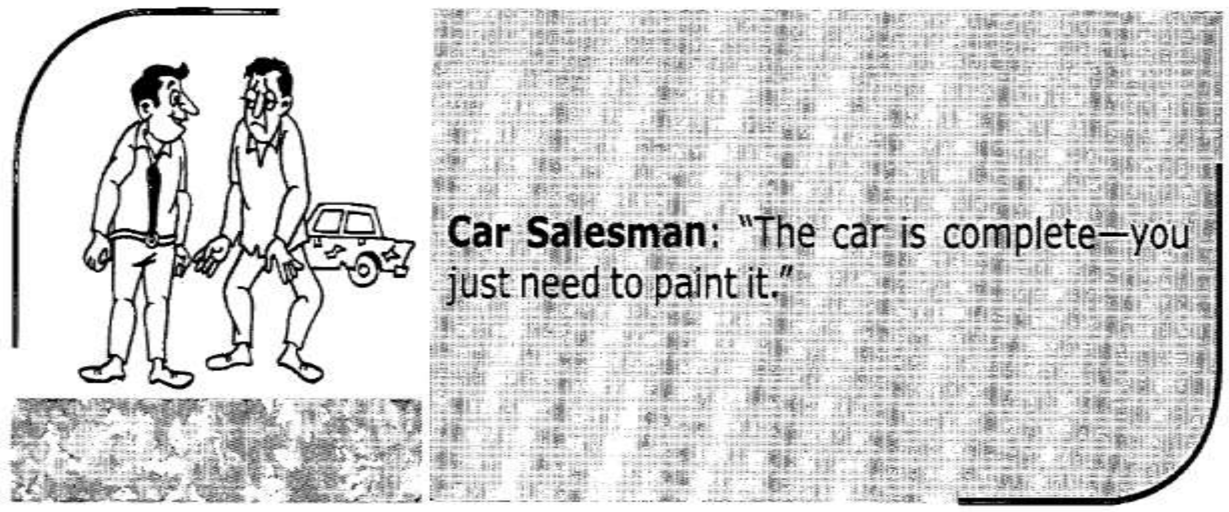
Figure 1.1 Relationship of effectiveness of testing to quality of other phases

If the quality of the other phases is low and the effectiveness of testing is low (lower left-hand corner of the grid), the situation is not sustainable. The product will most likely go out of business very soon. Trying to compensate for poor quality in other phases with increased emphasis on the testing phase (upper left-hand corner of the grid) is likely to put high pressure on everyone as the defects get detected closer to the time the product is about to be released. Similarly, blindly believing other phases to be of high quality and having a poor testing phase (lower right-hand side of the grid) will lead to the risky situation of unforeseen defects being detected at the last minute. The ideal state of course is when high quality is present in all the phases including testing (upper right-hand corner of the grid). In this state, the customers feel the benefits of quality and this promotes better teamwork and success in an organization.

1.1. The fundamental principles of testing are as follows.

1. The goal of testing is to find defects before customers find them out.
2. Exhaustive testing is not possible; program testing can only show the presence of defects, never their absence.
3. Testing applies all through the software life cycle and is not an end of-cycle activity.
4. Understand the reason behind the test.
5. Test the tests first.
6. Tests develop immunity and have to be revised constantly.
7. Defects occur in convoys or clusters, and testing should focus on these convoys.
8. Testing encompasses defect prevention.
9. Testing is a fine balance of defect prevention and defect detection.
10. Intelligent and well-planned automation is key to realizing the benefits of testing.
11. Testing requires talented, committed people who believe in themselves and work in teams.

1.2. THE INCOMPLETE CAR


Car Salesman: "The car is complete—you just need to paint it."

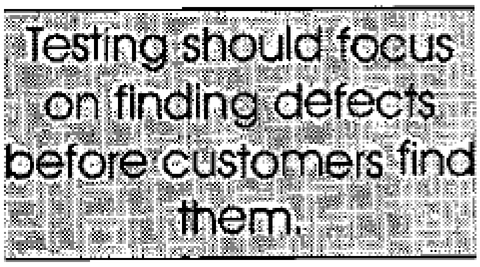
Eventually, whatever a software organization develops should meet the needs of the customer. Everything else is secondary. Testing is a means of making sure that the product meets the needs of the customer. We would like to assign a broader meaning to the term "customer." It does not mean just external customers. There are also internal customers. For example, if a product is built using different components from different groups within an organization, the users of these different components should be considered customers, even if they are from the same organization. Having this customer perspective enhances the quality of all the activities including testing.

We can take the internal customer concept a step further where the development team considers the testing team as its internal customer. This way we can ensure that the product is built not only for usage requirements but also for testing requirements. This concept improves "testability" of the product and improves interactions between the development and testing teams.

We would like to urge the reader to retain these two perspectives customer perspective and perspective of quality not being an add-on in the end, but built in every activity and component right from the beginning-throughout the life cycle.

If our job is to give a complete car to the customer (and not ask the customers to paint the car) and if our intent is to make sure the car works as expected, without any (major) problems, then we should ensure that we catch and correct all the defects in the car ourselves. This is the fundamental objective of testing. Anything we do in testing, it behaves us to remember that.

Sales representative / Engineer: "This car has the best possible transmission and brake, and accelerates from 0 to 80 mph in under 20 seconds!"
Customer: "Well, that may be true, but unfortunately it accelerates (even faster) when I press the brake pedal!"



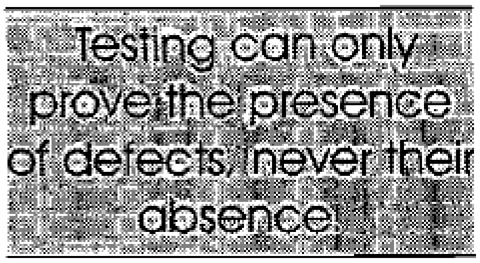
Testing should focus on finding defects before customers find them.

1.3. DIJKSTRA'S DOCTRINE

Consider a program that is supposed to accept a six-character code and ensure that the first character is numeric and rests of the characters are alphanumeric. **How many combinations of input data should we test, if our goal is to test the program exhaustively?**

The first character can be filled up in one of 10 ways (the digits 0-9).The second through sixth characters can each be filled up in 62 ways (digits 0-9, lower case letters a-z and capital letters A-Z). This means that we have a total of 10 x (62^5) or 91, 61, 32.832 valid combinations of values to test. Assuming that each combination takes 10 seconds to test, testing all these valid combinations will take approximately 2,905 years! Therefore, after 2,905 years, we may conclude that all valid inputs are accepted. But that is not the end of the story-what will happen to the program when we give invalid data? Continuing the above example, if we assume there are 10 punctuation characters, then we will have to spend a total of 44,176 years to test all the valid and invalid combinations of input data.

All this just to accept one field and test it exhaustively. Obviously, exhaustive testing of a real life program is never possible. All the above mean that we can choose to execute only a subset of the tests. To be effective, we should choose a subset of tests that can uncover the maximum number of errors.


Testing can only prove the presence of defects, never their absence.

1.4. A TEST IN TIME!

Defects in a product can come from any phase. There could have been errors while gathering initial requirements. If a wrong or incomplete requirement forms the basis for the design and development of a product, then that functionality can never be realized correctly in the eventual product. Similarly, when a product design-which forms the basis for the product development *(a La* coding)-is faulty, then the code that realizes the faulty design will also not meet the requirements. Thus, an essential condition should be that every phase of software development (requirements, design, coding, and so on) should catch and correct defects at that phase, without letting the defects seep to the next stage.

Let us look at the cost implications of letting defects seep through. If, during requirements capture, some requirements are erroneously captured and the error is not detected until the product is delivered to the customer, the organization incurs extra expenses for

- ❖ **Performing a wrong design based on the wrong requirements;**
- ❖ **transforming the wrong design into wrong code during the coding phase;**
- ❖ **testing to make sure the product complies with the (wrong) requirement; and**
- ❖ **Releasing the product with the wrong functionality.**

In **Figure 1.2** the defects in requirements are shown in gray. As you can see, these gray boxes are carried forward through three of the subsequent stages-design, coding and testing.

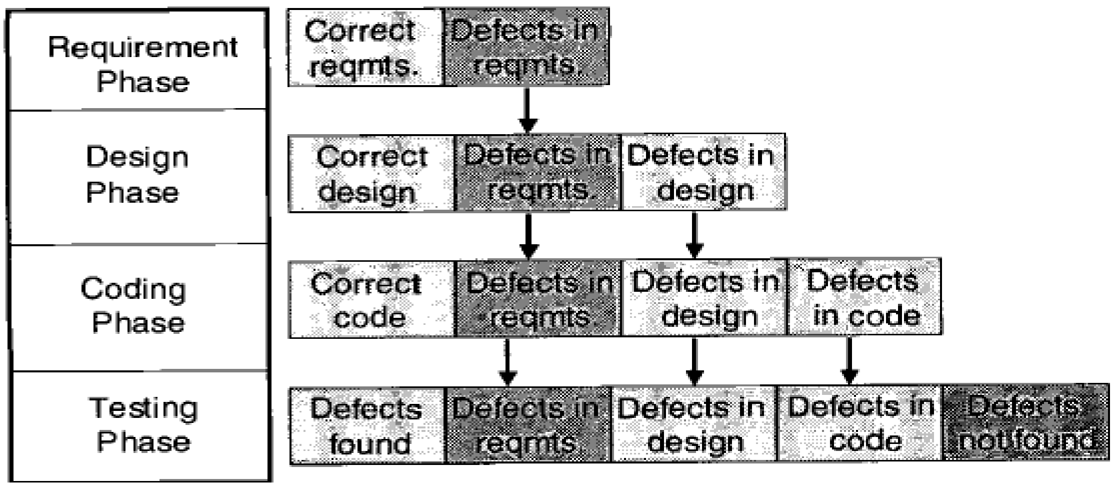


Figure 1.2: How defects from early phases add to the costs.

When this erroneous product reaches the customer after the testing phase, the customer may incur a potential downtime that can result in loss of productivity or business. **This in turn would reflect as a loss of goodwill to the software product organization. On top of this loss of goodwill, the software product organization would have to redo all the steps listed above, in order to rectify the problem.**

The cost of building a product and the number of defects in it increase steeply with the number of defects allowed to seep into the later phases.

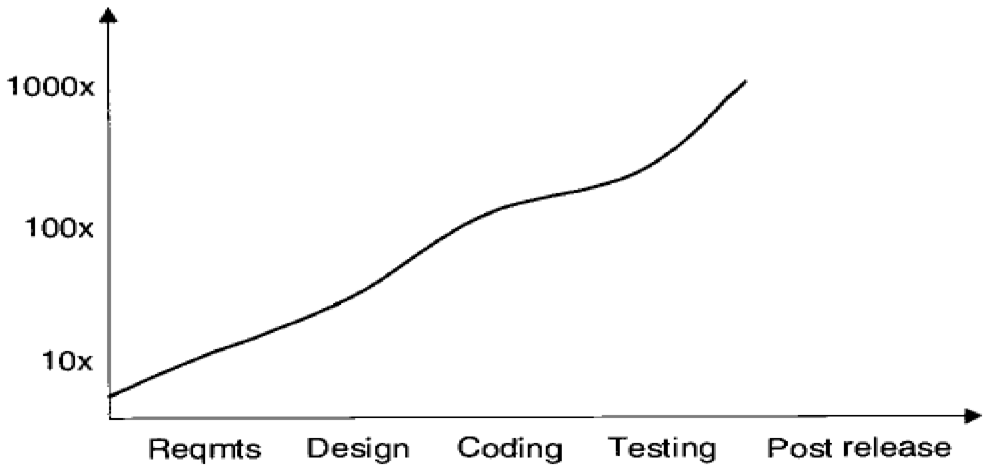
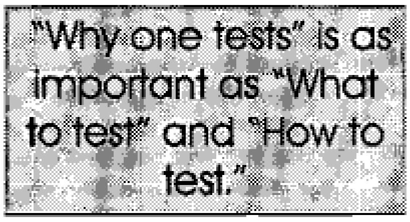


Figure 1.3: Compounding effects of defects on software costs

1.5. THE CAT AND THE SAINT

Testing requires asking about and understanding what you are trying to test, knowing what the correct outcome is, and why you are performing any test. If we carry out tests without understanding why we are running them, we will end up in running inappropriate tests that do not address what the product should do. In fact, it may even turn out that the product is modified to make sure the tests are run successfully, even if the product does not meet the intended customer needs!



"Why one tests" is as important as "What to test" and "How to test."

A saint sat meditating. A cat that was prowling around was disturbing his concentration. Hence he asked his disciples to tie the cat to a pillar while he meditated. This sequence of events became a daily routine. The tradition continued over the years with the saint's descendents and the cat's descendents. One day, there were no cats in the hermitage. The disciples got panicky and searched for a cat, saying, "*We need a cat. Only when we get a cat, can we tie it to a pillar and only after that can the saint start meditating!*"

1.6. TEST THE TESTS FIRST!

An audiologist was testing a patient, telling her, "I want to test the range within which you can hear. I will ask you from various distances to tell me your name, and you should tell me your name. Please turn back and answer." The patient understood what needs to be done.

Doctor (from 30 feet): What is your name?

...

Doctor (from 20 feet): What is your name?

...

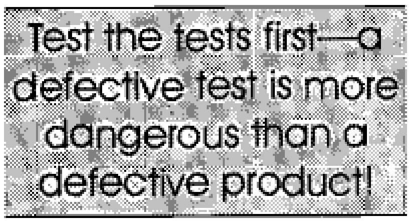
Doctor (from 10 feet): What is your name?

Patient: For the third time, let me repeat, my name is Sheela!

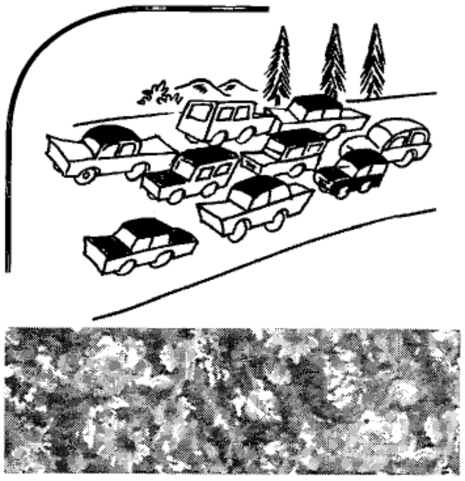
From the above example, it is clear that it is the audiologist **who has a hearing problem**, not the patient! Imagine if the doctor prescribed a treatment for the patient assuming that the latter could not hear at 20 feet and 30 feet. Tests are also artifacts produced by human beings, much as programs and documents are.

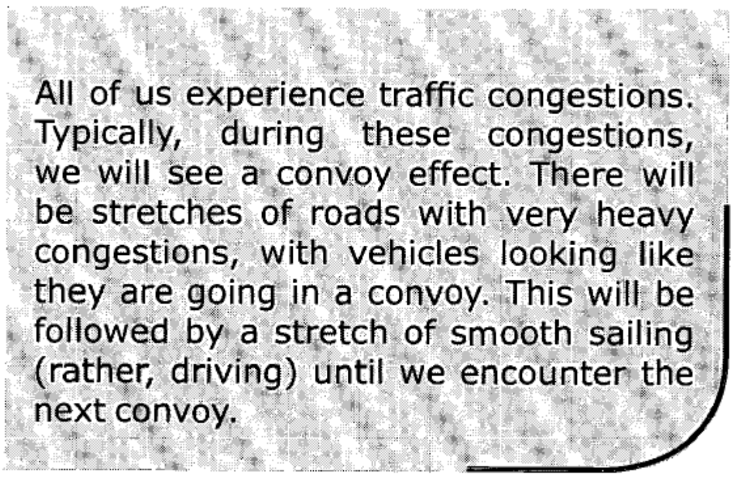
- **We cannot assume that the tests will be perfect either! It is important to make sure that the tests themselves are not faulty before we start using them.**
- **One way of making sure that the tests are tested is to document the inputs and expected outputs for a given test and have this description validated by an expert or get it counter-checked by some means outside the tests themselves.**

- For example, by giving a known input value and separately tracing out the path to be followed by the program or the process, one can manually ascertain the output that should be obtained.
- By comparing this **"known correct result"** with the result produced by the product, the confidence level of the test and the product can be increased.


Test the tests first—a defective test is more dangerous than a defective product!

1.7. THE CONVOY AND THE RAGS




All of us experience traffic congestions. Typically, during these congestions, we will see a convoy effect. There will be stretches of roads with very heavy congestions, with vehicles looking like they are going in a convoy. This will be followed by a stretch of smooth sailing (rather, driving) until we encounter the next convoy.

Defects in a program also typically display this convoy phenomenon. They occur in clusters. Glenford Myers, in his seminal work on software testing [MYER-79], proposed that the probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.

This may sound counter-intuitive, but can be logically reasoned out.

- ❖ A fix for one defect generally introduces some instability and necessitates another fix.
- ❖ All these fixes produce side-effects that eventually cause the convoy of defects in certain parts of the product.
- ❖ From a test planning perspective, this means that if we find defects in a particular part of product, more-not less-effort should be spent on testing that part.
- ❖ This will increase the return on investments in testing as the purpose of testing is finding the defects.

- This also means that whenever a product undergoes any change, these error-prone areas need to be tested as they may get affected.

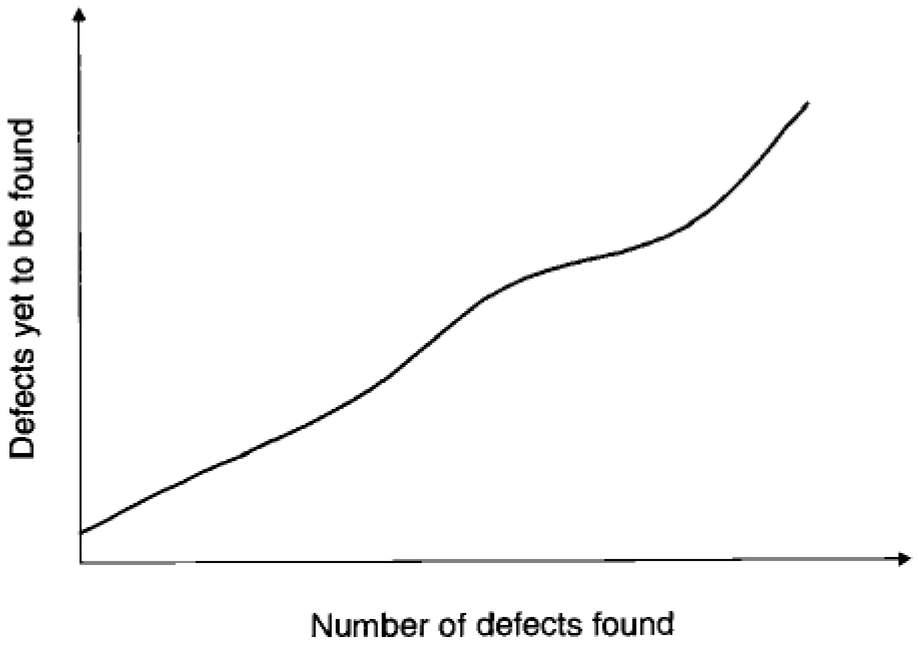
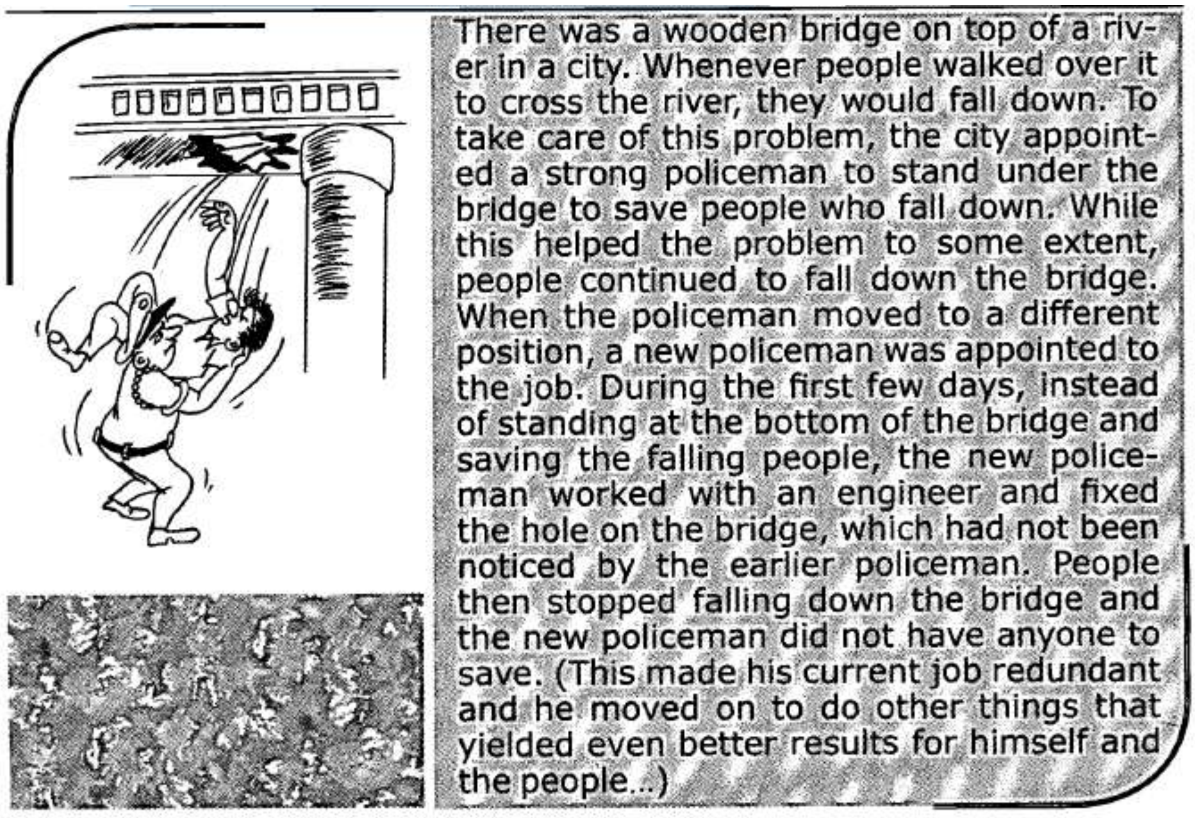


Figure 1.4: The number of defects yet to be found increases with the number of defects uncovered.

A fix for a defect is made around certain lines of code. This fix can produce side-effects around the same piece of code. This sets in spiraling changes to the program, all localized to certain select portions of the code. When we look at the code that got the fixes for the convoy of defects, it is likely to look like a piece of rag! **Fixing a tear in one place in a shirt would most likely cause damage in another place. The only long-term solution in such a case is to throwaway the shirt and creates a new one. This amounts to a re-architecting the design and rewriting the code.**

1.8. THE POLICEMEN ON THE BRIDGE



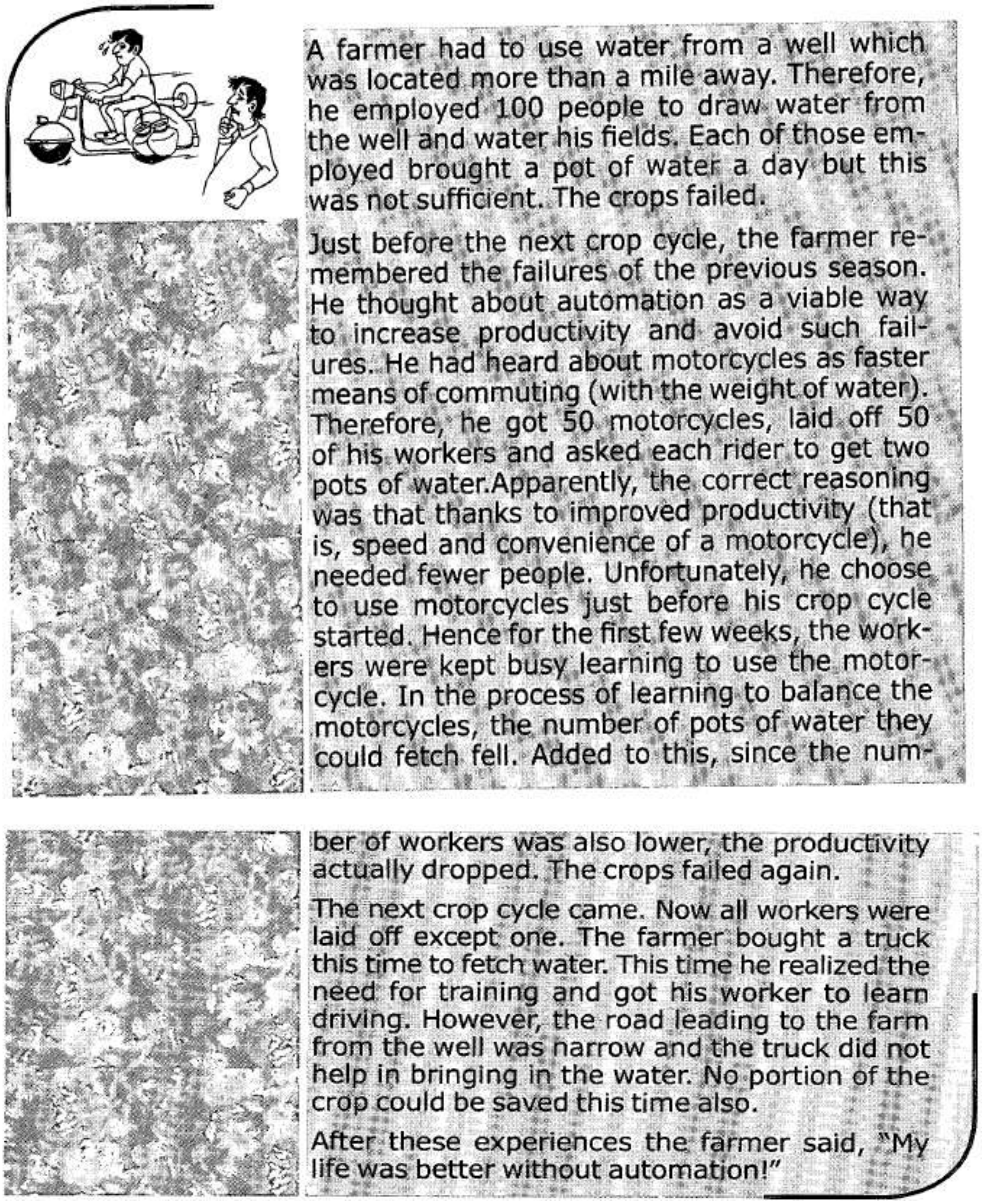
There was a wooden bridge on top of a river in a city. Whenever people walked over it to cross the river, they would fall down. To take care of this problem, the city appointed a strong policeman to stand under the bridge to save people who fall down. While this helped the problem to some extent, people continued to fall down the bridge. When the policeman moved to a different position, a new policeman was appointed to the job. During the first few days, instead of standing at the bottom of the bridge and saving the falling people, the new policeman worked with an engineer and fixed the hole on the bridge, which had not been noticed by the earlier policeman. People then stopped falling down the bridge and the new policeman did not have anyone to save. (This made his current job redundant and he moved on to do other things that yielded even better results for himself and the people...)

Testers are probably best equipped to know the problems customers may encounter. Like the second police officer in the above story, they know people fall and they know why people fall. Rather than simply catch people who fall (and thereby be exposed to the risk of a missed catch), they should also look at the root cause for falling and advise preventive action. It may not be possible for testers themselves to carry out preventive action. Just as the second police officer had to enlist the help of an engineer to plug the hole, testers would have to work with development engineers to make sure the root cause of the defects are addressed. The testers should not feel that by eliminating the problems totally their jobs are at stake. Like the second policeman, their careers can be enriching and beneficial to the organization if they harness their defect detection experience and transform some of it to defect prevention initiatives.

- **Defect prevention is a part of a tester's job.**
- A career as a tester can be enriching and rewarding, if we can balance defect prevention and defect detection activities.

Prevention is better than cure—you may be able to expand your horizon much farther.

1.9. AUTOMATION SYNDROME



A farmer had to use water from a well which was located more than a mile away. Therefore, he employed 100 people to draw water from the well and water his fields. Each of those employed brought a pot of water a day but this was not sufficient. The crops failed.

Just before the next crop cycle, the farmer remembered the failures of the previous season. He thought about automation as a viable way to increase productivity and avoid such failures. He had heard about motorcycles as faster means of commuting (with the weight of water). Therefore, he got 50 motorcycles, laid off 50 of his workers and asked each rider to get two pots of water.Apparently, the correct reasoning was that thanks to improved productivity (that is, speed and convenience of a motorcycle), he needed fewer people. Unfortunately, he choose to use motorcycles just before his crop cycle started. Hence for the first few weeks, the workers were kept busy learning to use the motorcycle. In the process of learning to balance the motorcycles, the number of pots of water they could fetch fell. Added to this, since the number of workers was also lower, the productivity actually dropped. The crops failed again.

The next crop cycle came. Now all workers were laid off except one. The farmer bought a truck this time to fetch water. This time he realized the need for training and got his worker to learn driving. However, the road leading to the farm from the well was narrow and the truck did not help in bringing in the water. No portion of the crop could be saved this time also.

After these experiences the farmer said, "My life was better without automation!"

If you go through the story closely there appear **to be several reasons for the crop failures** that are not to do with the automation intent at all.

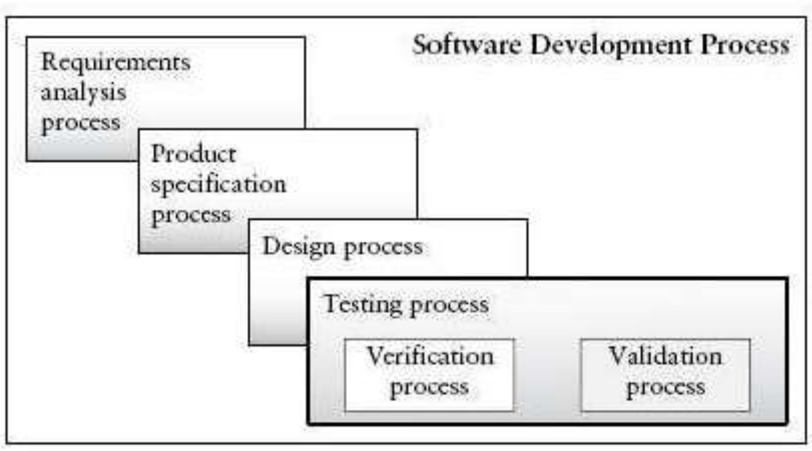
- ❖ The frustration of the farmer should not be directed at automation but on the process followed for automation and the inappropriate choices made.

- In the second crop cycle, the reason for failure was lack of skills and in the third cycle it is due to improper tool implementation.
- In the first crop cycle, the farmer laid off his workers immediately after the purchase of motorcycles and expected cost and time to come down.
- He repeated the same mistake for the third crop cycle. Automation does not yield results immediately.
- The moral of the above story as it applies to testing is that automation requires careful planning, evaluation, and training.
- Automation may not produce immediate returns.
- An organization that expects immediate returns from automation may end up being disappointed and wrongly blame automation for their failures, instead of objectively looking at **their level of preparedness for automation in terms of planning, evaluation, and training.**

1.10. TESTING AS A PROCESS

The software development process has been described as a series of phases, procedures, and steps that result in the production of a software product. Embedded within the software development process are several other processes including testing. Some of these are shown in **Figure 1.5.** Testing itself is related to two other processes called verification and validation as shown in **Figure 1.5.**

- **Validation is the process of evaluating a software system or component during, or at the end of, the development cycle in order to determine whether it satisfies specified requirements.**
- **Verification is the process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.**



Software Development Process
Requirements analysis process
Product specification process
Design process
Testing process
Verification process
Validation process

Figure 1.5: Example processes embedded in the software development process

- ❖ **Verification is usually associated with activities such as inspections and reviews of software deliverables.**
- ❖ **Testing itself has been defined in several ways. Two definitions are shown below.**
    1. **Testing is generally described as a group of procedures carried out to evaluate some aspect of a piece of software.**
    2. **Testing can be described as a process used for revealing defects in software, and for establishing that the software has attained a specified degree of quality with respect to selected attributes.**
- ❖ **Debugging, or fault localization is the process of (1) locating the fault or defect, (2) repairing the code, and (3) retesting the code.**

1.11. Basic definitions:

Errors

An error is a mistake, misconception, or misunderstanding on the part of a software developer. In the category of developer we include software engineers, programmers, analysts, and testers. **For example, a developer may misunderstand a design notation, or a programmer might type a variable name incorrectly.**

Faults (Defects)

A fault (defect) is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification.

Faults or defects are sometimes called —bugs.

Use of the latter term trivializes the impact faults have on software quality. Use of the term —defect is also associated with software artifacts such as requirements and design documents. Defects occurring in these artifacts are also caused by errors and are usually detected in the review process.

Failures

A failure is the inability of a software system or component to perform its required functions within specified performance requirements**.**

During execution of a software component or system, a tester, developer, or user observes that it does not produce the expected results.

Test case

A test case in a practical sense is a test-related item which contains the following information:

1. **A set of test inputs.** These are data items received from an external source by the code under test. The external source can be hardware, software, or human.
2. **Execution conditions.** These are conditions required for running the test, for example, a certain state of a database, or a configuration of a hardware device.
3. **Expected outputs.** These are the specified results to be produced by the code under test.

Test

A test is a group of related test cases, or a group of related test cases and test procedures**.**

Test Oracle

A test oracle is a document, or piece of software that allows testers to determine whether a test has been passed or failed.

A program, or a document that produces or specifies the expected outcome of a test, can serve as an oracle. Examples include a specification (especially one that contains pre- and post conditions), a design document, and a set of requirements. Other sources are regression test suites. The suites usually contain components with correct results for previous versions of the software. If some of the functionality in the new version overlaps the old version, the appropriate oracle information can be extracted. A working trusted program can serve as its own oracle in a situation where it is being ported to a new environment. In this case its intended behavior should not change in the new environment.

Test Bed

A test bed is an environment that contains all the hardware and software needed to test a software component or a software system. This includes the entire testing environment, for example, simulators, emulators, memory checkers, hardware probes, software tools, and all other items needed to support execution of the tests.

Software Quality

1. Quality relates to the degree to which a system, system component, or process meets specified requirements.

2. Quality relates to the degree to which a system, system component, or process meets customer or user needs, or expectations.

In order to determine whether a system, system component, or process is of high quality we use what are called quality attributes. the degree to which they possess a given quality attribute with quality metrics.

Quality metric

A metric is a quantitative measure of the degree to which a system, system component, or process possesses a given attribute.

There are product and process metrics. A very commonly used example of a software product metric is software size, usually measured in lines of code (LOC). Two examples of commonly used process metrics are costs and time required for a given task. Quality metrics are a special kind of metric.

A quality metric is a quantitative measurement of the degree to which an item possesses a given quality attribute.

Some examples of quality attributes with brief explanations are the following:

- ❖ **Correctness** —the degree to which the system performs its intended function.
- ❖ **Reliability** —the degree to which the software is expected to perform its required functions under stated conditions for a stated period of time.
- ❖ **Usability** —relates to the degree of effort needed to learn, operate, prepare input, and interpret output of the software.
- ❖ **Integrity** —relates to the system‘s ability to withstand both intentional and accidental attacks.
- ❖ **Portability—**relates to the ability of the software to be transferred from one environment to another.

- **Maintainability—**the effort needed to make changes in the software.
- **Interoperability—**the effort needed to link or couple one system to another.

Software Quality Assurance Group

The software quality assurance (SQA) group in an organization has ties to quality issues. The group serves as the customer's representative and advocate. Their responsibility is to look after the customers interests.

The software quality assurance (SQA) group is a team of people with the necessary training and skills to ensure that all necessary actions are taken during the development process so that the resulting software conforms to established technical requirements.

Review

A review is a group meeting whose purpose is to evaluate a software artifact or a set of software artifacts.

The composition of a review group may consist of managers, clients, developers, testers and other personnel depending on the type of artifact under review. A special type of review called an audit is usually conducted by a Software Quality Assurance group for the purpose of assessing compliance with specifications, and/or standards, and/or contractual agreements.

1.12. Origins of defects

The term ***defect*** and its relationship to the terms ***error*** **and** ***failure*** in the context of the software development domain. Defects have detrimental effects on software users, and software engineers work very hard to produce high-quality software with a low number of defects. But even under the best of development circumstances errors are made, resulting in defects being injected in the software during the phases of the software life cycle. Defects as shown in **Figure 1.6** stem from the following sources:

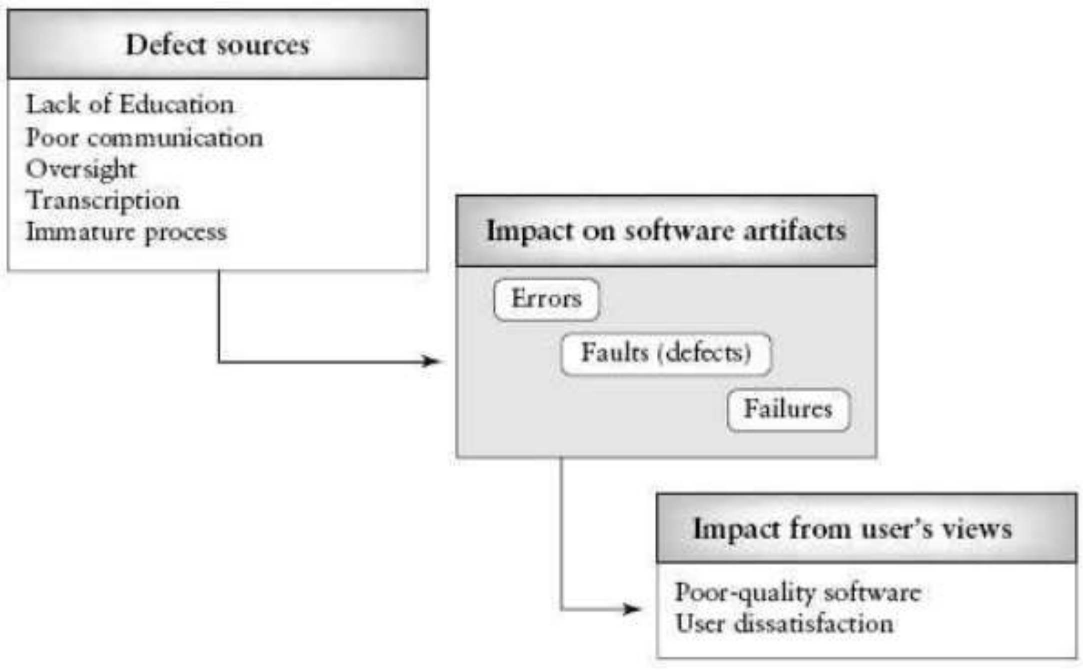


Figure 1.6: Origins of defects

1. ***Education:*** The software engineer did not have the proper educational background to prepare the software artifact. He/She did not understand how to do something. For example, a software engineer who did not understand the precedence order of operators in a particular programming language could inject a defect in an equation that uses the operators for a calculation.

2. ***Communication:*** The software engineer was not informed about something by a colleague. For example, if engineer 1 and engineer 2 are working on interfacing modules, and engineer 1 does not inform engineer 2 that a no error checking code will appear in the interfacing module he is developing, engineer 2 might make an incorrect assumption relating to the presence/absence of an error check, and a defect will result.

3. ***Oversight:*** The software engineer omitted to do something. For example, a software engineer might omit an initialization statement.

4. ***Transcription:*** The software engineer knows what to do, but makes a mistake in doing it. A simple example is a variable name being misspelled when entering the code.

5. ***Process:*** The process used by the software engineer misdirected her actions. For example, a development process that did not allow sufficient time for a detailed specification to be developed and reviewed could lead to specification defects.

1.13. Developer/Tester Support for Developing a Defect Repository

The focus is to show with examples some of the most common types of defects that occur during software development. It is important if you are a member of a test organization to illustrate to management and your colleagues the benefits of developing a defect repository to store defect information.

As software engineers and test specialists we should follow the examples of engineers in other disciplines who have realized the usefulness of defect data. A requirement for repository development should be a part of testing and/or debugging policy statements. You begin with development of a defect classification scheme and then initiate the collection defect data from organizational projects. Forms and templates will need to be designed to collect the data.

We will need to be conscientious about recording each defect after testing, and also recording the frequency of occurrence for each of the defect types. Defect monitoring should continue for each on-going project. The distribution of defects will change as you make changes in your processes. The defect data is useful for test planning, a **Testing Maturity Model** (TMM) level 2 maturity goals. It helps you to select applicable testing techniques, design (and reuse) the test cases you need, and allocate the amount of resources you will need to devote to detecting and removing these defects. This in turn will allow you to estimate testing schedules and costs.

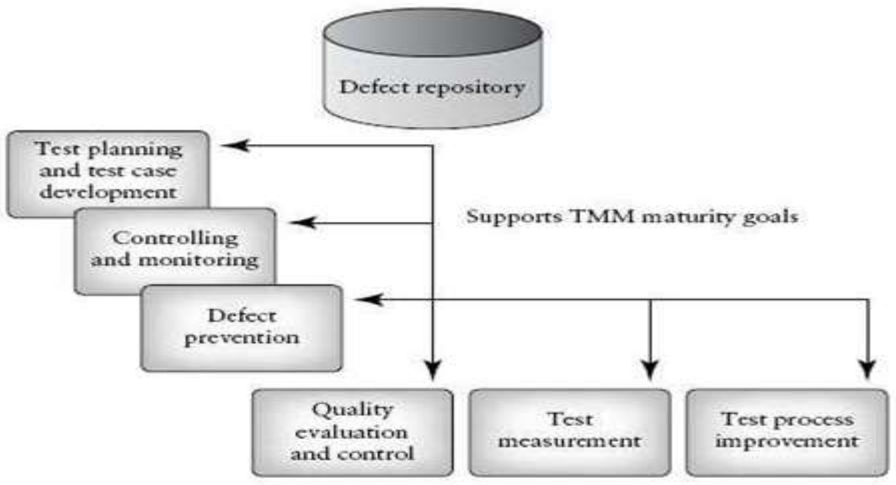

Defect repository
Test planning and test case development
Controlling and monitoring
Defect prevention
Supports TMM maturity goals
Quality evaluation and control
Test measurement
Test process improvement

Figure 1.7: The Defect repository, and support for TMM (Testing Maturity Model) maturity goals

The defect data can support debugging activities as well. In fact, as **Figure 1.7** shows, a defect repository can help to support achievement and continuous implementation of several TMM maturity goals including controlling and monitoring of test, software quality evaluation and control, test measurement, and test process improvement.

****