The failure here is the incorrect interest calculations for customers. The defect is the wrong calculation in the code. The root cause was the product owner's lack of knowledge about how interest should be calculated, and the effect was customer complaints.

The root cause can be addressed by providing additional training in interest rate calculations to the product owner, and possibly additional reviews of user stories by interest calculation experts. If this is done, then incorrect interest calculations due to ambiguous user stories should be a thing of the past.

Root cause analysis is covered in more detail in two other ISTQB qualifications: Expert Level Test Management, and Expert Level Improving the Test Process.

1.3 SEVEN TESTING PRINCIPLES

SYLLABUS LEARNING OBJECTIVES FOR 1.3 SEVEN TESTING PRINCIPLES (K2)

FL-1.3.1 Explain the seven testing principles (K2)

In this section, we will review seven fundamental principles of testing that have been observed over the last 40+ years. These principles, while not always understood or noticed, are in action on most if not all projects. Knowing how to spot these principles, and how to take advantage of them, will make you a better tester.

In addition to the descriptions of each principle below, you can refer to Table 1.1 for a quick reference of the principles and their text as written in the Syllabus.

Principle 1. Testing shows the presence of defects, not their absence

As mentioned in the previous section, a typical objective of many testing efforts is to find defects. Many testing organizations that the authors have worked with are quite effective at doing so. One of our exceptional clients consistently finds, on average, 99.5% of the defects in the software it tests. In addition, the defects left undiscovered are less important and unlikely to happen frequently in production. Sometimes, it turns out that this test team has indeed found 100% of the defects that would matter to customers, as no previously unreported defects are reported after release. Unfortunately, this level of effectiveness is not common.

However, no test team, test technique or test strategy can guarantee to achieve 100% defect-detection percentage (DDP) – or even 95%, which is considered excellent. Thus, it is important to understand that, while testing can show that defects are present, it cannot prove that there are no defects left undiscovered. Of course, as testing continues, we reduce the likelihood of defects that remain undiscovered, but eventually a form of Zeno's paradox takes hold: each additional test run may cut the risk of a remaining defect in half, but only an infinite number of tests can cut the risk down to zero.

That said, testers should not despair or let the perfect be the enemy of the good. While testing can never prove that the software works, it can reduce the remaining level of risk to product quality to an acceptable level, as mentioned before. In any endeavour worth doing, there is some risk. Software projects – and software testing – are endeavours worth doing.

TABLE 1.1 Testing principles

Principle 1:	**Testing shows the presence of defects, not their absence**	Testing can show that defects are present, but cannot prove that there are no defects. Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found, testing is not a proof of correctness.
Principle 2:	**Exhaustive testing is impossible**	Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases. Rather than attempting to test exhaustively, risk analysis, test techniques and priorities should be used to focus test efforts.
Principle 3:	**Early testing saves time and money**	To find defects early, both static and dynamic test activities should be started as early as possible in the software development life cycle. Early testing is sometimes referred to as 'shift left'. Testing early in the software development life cycle helps reduce or eliminate costly changes (see Chapter 3, Section 3.1).
Principle 4:	**Defects cluster together**	A small number of modules usually contains most of the defects discovered during pre-release testing, or they are responsible for most of the operational failures. Predicted defect clusters, and the actual observed defect clusters in test or operation, are an important input into a risk analysis used to focus the test effort (as mentioned in Principle 2).
Principle 5:	**Beware of the pesticide paradox**	If the same tests are repeated over and over again, eventually these tests no longer find any new defects. To detect new defects, existing tests and test data are changed and new tests need to be written. (Tests are no longer effective at finding defects, just as pesticides are no longer effective at killing insects after a while.) In some cases, such as automated regression testing, the pesticide paradox has a beneficial outcome, which is the relatively low number of regression defects.
Principle 6:	**Testing is context dependent**	Testing is done differently in different contexts. For example, safety-critical software is tested differently from an e-commerce mobile app. As another example, testing in an Agile project is done differently to testing in a sequential life cycle project (see Chapter 2, Section 2.1).
Principle 7:	**Absence-of-errors is a fallacy**	Some organizations expect that testers can run all possible tests and find all possible defects, but Principles 2 and 1, respectively, tell us that this is impossible. Further, it is a fallacy to expect that *just* finding and fixing a large number of defects will ensure the success of a system. For example, thoroughly testing all specified requirements and fixing all defects found could still produce a system that is difficult to use, that does not fulfil the users' needs and expectations or that is inferior compared to other competing systems.

Principle 2. Exhaustive testing is impossible

This principle is closely related to the previous principle. For any real-sized system (anything beyond the trivial software constructed in first-year software engineering courses), the number of possible test cases is either infinite or so close to infinite as to be practically innumerable.

Infinity is a tough concept for the human brain to comprehend or accept, so let's use an example. One of our clients mentioned that they had calculated the number of possible internal data value combinations in the Unix operating system as greater than the number of known molecules in the universe by four orders of magnitude. They further calculated that, even with their fastest automated tests, just to test all of these internal state combinations would require more time than the current age of the universe. Even that would not be a complete test of the operating system; it would only cover all the possible data value combinations.

So, we are confronted with a big, infinite cloud of possible tests; we must select a subset from it. One way to select tests is to wander aimlessly in the cloud of tests, selecting at random until we run out of time. While there is a place for automated random testing, by itself it is a poor strategy. We'll discuss testing strategies further in Chapter 5, but for the moment let's look at two.

One strategy for selecting tests is risk-based testing. In risk-based testing, we have a cross-functional team of project and product stakeholders perform a special type of risk analysis. In this analysis, stakeholders identify risks to the quality of the system, and assess the level of risk (often using likelihood and impact) associated with each risk item. We focus the test effort based on the level of risk, using the level of risk to determine the appropriate number of test cases for each risk item, and also to sequence the test cases.

Another strategy for selecting tests is requirements-based testing. In requirements-based testing, testers analyze the requirements specification (which would be user stories in Agile projects) to identify test conditions. These test conditions inherit the priority of the requirement or user story they derive from. We focus the test effort based on the priority to determine the appropriate number of test cases for each aspect, and also to sequence the test cases.

Principle 3. Early testing saves time and money

This principle tells us that we should start testing as early as possible in order to find as many defects as possible. In addition, since the cost of finding and removing a defect increases the longer that defect is in the system, early testing also means we are likely to minimize the cost of removing defects.

So, the first principle tells us that we cannot find all the bugs, but rather can only find some percentage of them. The second principle tells us that we cannot run every possible test. The third principle tells us to start testing early. What can we conclude when we put these three principles together?

Imagine that you have a system with 1,000 defects. Suppose we wait until the very end of the project and run one level of testing, system test. You find and fix 90% of the defects. That still leaves 100 defects, which presumably will escape to the customers or users.

Instead, suppose that you start testing early and continue throughout the life cycle. You perform requirements reviews, design reviews and code reviews. You perform unit testing, integration testing and system testing. Suppose that, during each test activity, you find and remove only 45% of the defects – half as effective

as the previous system test level. Nevertheless, at the end of the process, fewer than 30 defects remain. Even though each test activity was only 45% effective at finding defects, the overall sequence of activities was 97% effective. Note that now we are doing both static testing (the reviews) and dynamic testing (the running of tests at the different test levels). This approach of starting test activities as early as possible is also called 'shift left' because the test activities are no longer all done on the right-hand side of a sequential life cycle diagram, but on the left-hand side at the beginning of development. Although unit test execution is of course on the right side of a sequential life cycle diagram, improving and spending more effort on unit testing early on is a very important part of the shift left paradigm.

In addition, defects removed early cost less to remove. Further, since much of the cost in software engineering is associated with human effort, and since the size of a project team is relatively inflexible once that project is underway, reduced cost of defects also means reduced duration of the project. That situation is shown graphically in Figure 1.3.

Now, this type of cumulative and highly efficient defect removal only works if each of the test activities in the sequence is focused on different, defined objectives. If we simply test the same test conditions over and over, we will not achieve the cumulative effect, for reasons we will discuss in a moment.

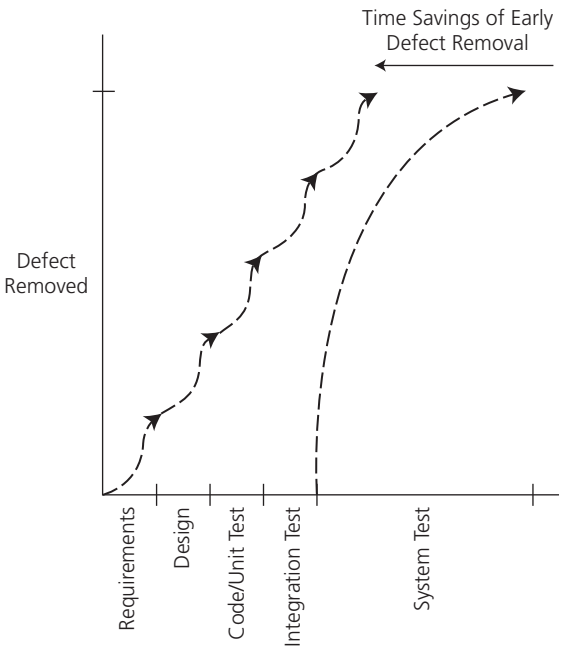

FIGURE 1.3 Time savings of early defect removal

Principle 4. Defects cluster together

This principle relates to something we discussed previously, that relying entirely on the testing strategy of a random walk in the infinite cloud of possible tests is relatively weak. Defects are not randomly and uniformly distributed throughout the software under test. Rather, defects tend to be found in clusters, with 20% (or fewer)

of the modules accounting for 80% (or more) of the defects. In other words, the defect density of modules varies considerably. While controversy exists about why defect clustering happens, the reality of defect clustering is well established. It was first demonstrated in studies performed by IBM in the 1960s [Jones 2008], and is mentioned in Myers [2011]. We continue to see evidence of defect clustering in our work with clients.

Defect clustering is helpful to us as testers, because it provides a useful guide. If we focus our test effort (at least in part) based on the expected (and ultimately observed) likelihood of finding a defect in a certain area, we can make our testing more effective and efficient, at least in terms of our objective of finding defects. Knowledge of and predictions about defect clusters are important inputs to the risk-based testing strategy discussed earlier. In a metaphorical way, we can imagine that bugs are social creatures who like to hang out together in the dark corners of the software.

Principle 5. Beware of the pesticide paradox

This principle was coined by Boris Beizer [Beizer 1990]. He observed that, just as a pesticide repeatedly sprayed on a field will kill fewer and fewer bugs each time it is used, so too a given set of tests will eventually stop finding new defects when re-run against a system under development or maintenance. If the tests do not provide adequate coverage, this slowdown in defect finding will result in a false level of confidence and excessive optimism among the project team. However, the air will be let out of the balloon once the system is released to customers and users.

Using the right test strategies is the first step towards achieving adequate coverage. However, no strategy is perfect. You should plan to regularly review the test results during the project, and revise the tests based on your findings. In some cases, you need to write new and different tests to exercise different parts of the software or system. These new tests can lead to discovery of previously unknown defect clusters, which is a good reason not to wait until the end of the test effort to review your test results and evaluate the adequacy of test coverage.

The pesticide paradox is important when implementing the multilevel testing discussed previously in regards to the principle of early testing. Simply repeating our tests of the same conditions over and over will not result in good cumulative defect detection. However, when used properly, each type and level of testing has its own strengths and weaknesses in terms of defect detection, and collectively we can assemble a very effective sequence of defect filters from them. After such a sequence of complementary test activities, we can be confident that the coverage is adequate, and that the remaining level of risk is acceptable.

Sometimes the pesticide paradox can work in our favour, if it is not *new* defects that we are looking for. When we run automated regression tests, we are ensuring that the software that we are testing is still working as it was before; that is, there are no new unexpected side-effect defects that have appeared as a result of a change elsewhere. In this case, we are pleased that we have not found any new defects.

Principle 6. Testing is context dependent

Our safety-critical clients test with a great deal of rigour and care – and cost. When lives are at stake, we must be extremely careful to minimize the risk of undetected defects. Our clients who release software on the web, such as e-commerce sites, or who develop mobile apps, can take advantage of the possibility to quickly change the software when necessary, leading to a different set of testing challenges – and opportunities. If you tried to apply safety-critical approaches to a mobile app, you