1 Introduction to Software Testing

This chapter discusses the motivations for testing software, and also discusses why exhaustive testing is not generally feasible, and thus various test heuristics must be used. These test heuristics, and the lack of a standard software specification[1] language, are what makes software testing as much an art as a science.

1.1 The Software Industry

The software industry has come a long way since its beginnings in the 1950s. The independent software industry was essentially born in 1969 when IBM announced that it would stop treating its software as a free add-on to its computers, and instead would sell software and hardware as separate products. This opened up the market to external companies that could produce and sell software for IBM machines.

Software products for mass consumption arrived in 1981 with the launch of PC-based software packages. Another dramatic boost came in the 1990s with the arrival of the World Wide Web, and in the 2000s with mobile devices. In 2010 the top 500 companies in the global software industry had revenues of \$492 billion, and by 2018 this had risen to \$868 billion.[2] The industry is extremely dynamic and continually undergoing rapid change as new innovations appear. Unlike some other industries, for example transportation, it is still in many ways an immature industry. It does not, in general, have a set of quality standards that have been gained through years of experience.

Numerous examples exist of the results of failures in software quality and the costs it can incur. Well-publicised incidents include the failure of the European Space Agency's Ariane 5 rocket, the Therac-25 radiation therapy machine, and the loss of the Mars Climate Orbiter in 1999. A study by the US Department of Commerce's National Institute of Standards and Technology (NIST) in 2002 estimated that the annual cost of inadequate software testing to the US economy was up to \$59.5 billion per year.[3]

[1] A software specification defines clearly and unambiguously what the software must do.
[2] *Software Magazine*. 2018 Software 500 Companies. Available at: www.rcpbuyersguide.com/top-companies.php.
[3] NIST. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. NIST, 2002.

However, many participants in the industry do apply quality models and measures to the processes through which their software is produced. Software testing is an important part of the software quality assurance process, and is an important discipline within software engineering. It has an important role to play throughout the software development life cycle, whether being used in a *verification* and *validation* context, or as part of a test-driven software development process such as *eXtreme Programming*.

Software engineering as a discipline grew out of the *software crisis*. This term was first used at the end of the 1960s, but it really began to have meaning through the 1970s as the software industry was growing. This reflected the increasing size and complexity of software projects combined with the lack of formal procedures for managing such projects. This resulted in a number of problems:

- projects were running over budget;
- projects were running over time;
- the software products were of low quality;
- the software products often did not meet requirements;
- projects were chaotic; and
- software maintenance was increasingly difficult.

If the software industry was to keep growing, and the use of software was to become more widespread, this situation could not continue. The solution was to formalise the roles and responsibilities of software engineering personnel. These software engineers would plan and document in detail the goals of each software project and how it was to be carried out; they would manage the process via which the software code would be created; and they would ensure that the end result had attributes that showed it was a quality product. This relationship between quality management and software engineering meant that software testing would be integrated into its field of influence. Moreover, the field of software testing was also going to have to change if the industry wanted to get over the software crisis.

While the difference between debugging a program and testing a program was recognised by the 1970s, it was only from this time on that testing began to take a significant role in the production of software. It was to change from being an activity that happened at the end of the product cycle, to check that the product worked, to an activity that takes place throughout each stage of development, catching faults as early as possible. A number of studies comparing the relative costs of early and late defect detection have all reached the same conclusion: the earlier the defect is caught, the lower the cost of fixing it.

The progressive improvement of software engineering practices has led to a significant improvement in software quality. The short-term benefits of software testing to the business include improving the performance, interoperability and conformance of the software products produced. In the longer term, testing reduces the future costs, and builds customer confidence.

Software testing is integrated into many of the software development processes in use today. Approaches such as *test driven development* (TDD) use testing to drive the code development. In this approach, the tests are developed (often with the assistance of the end-user or customer) before the code is written.

1.1.1 Software Testing and Quality

Proper software testing procedures reduce the risks associated with software development. Modern programs are often very complex, having millions of lines of code and multiple interactions with other software systems. And they often implement a solution that has been defined in very abstract terms, described as a vague set of requirements lacking in exactness and detail. Quality problems are further compounded by external pressures on developers from business owners, imposing strict deadlines and budgets to reduce both the time to market and associated production costs. These pressures can result in inadequate software testing, leading to reduced quality. Poor quality leads to increased software failures, increased development costs, and increased delays in releasing software. More severe outcomes for a business can be a loss of reputation, leading to reduced market share, or even to legal claims.

The international standard ISO/IEC 25010[4] defines a product quality model with eight quality characteristics (Table 1.1).

Table 1.1 Software quality attributes in ISO/IEC 25010.

Attribute	Characteristics
Functional suitability	Functional completeness, functional correctness, functional appropriateness
Performance efficiency	Time behaviour, resource utilisation, capacity
Compatibility	Coexistence, interoperability
Usability	Appropriateness, recognisability, learnability, operability, user error protection, user interface aesthetics, accessibility
Reliability	Maturity, availability, fault tolerance, recoverability
Security	Confidentiality, integrity, non-repudiation, authenticity, accountability
Maintainability	Modularity, reusability, analysability, modifiability, testability
Portability	Adaptability, installability, conformance, replaceability

Attributes that can be measured objectively, such as performance and functionality, are easier to test than those that require a subjective opinion, such as learnability and installability.

1.1.2 Software Testing and Risk Management

Software testing can be viewed as a risk-management activity. The more resources that are spent on testing, the lower the probability of a software failure, but the higher

[4] ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models.

the testing cost. One factor in risk management is to compare the expected cost of failure against the cost of testing. The expected cost is estimated as follows:

$$\text{expected cost} = \text{risk of failure} \times \text{cost of failure}$$

For a business, there are short- and long-term costs associated with failure. The short-term costs are primarily related to fixing the problem, but may also be from lost revenue if product release is delayed. The long-term costs are primarily the costs of losing reputation and associated sales.

The cost of testing needs to be proportional to income and the cost of failure (with the current state of the art, it is arguable that all software is subject to failure at some stage). The effectiveness of testing can generally be increased by the testing team being involved early in the process. Direct involvement of customers/users is also an effective strategy. The expected cost of failure is controlled through reducing the probability of failure through rigorous engineering development practices and quality assurance (testing is part of the quality assurance process).

Software testing can be addressed as an optimisation process: getting the best return for the investment. Increased investment in testing reduces the cost of software failures, but increases the cost of software development. The key is to find the best balance between these costs. This interaction between cost of testing and profit is demonstrated in Figure 1.1.

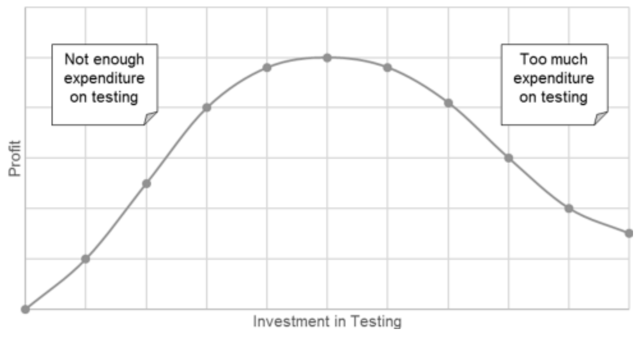

Figure 1.1 Cost of testing vs. profit.

1.2 Mistakes, Faults, and Failures

Leaving aside the broader relationship between software testing and the attributes of quality for the moment, the most common application of testing is to search for defects present in a piece of software and/or verify that particular defects are not present in that software. The term software defect is often expanded into three categories: mistakes, faults, and failures.

1. **Mistakes**: these are made by software developers. These are conceptual errors and can result in one or more faults in the source code.

2. **Faults**: these are flaws in the source code, and can be the product of one or more mistakes. Faults can lead to failures during program execution.
3. **Failures**: these are symptoms of a fault, and consist of incorrect, or out-of-specification behaviour by the software. Faults may remain hidden until a certain set of conditions are met which reveal them as a failure in the software execution. When a failure is detected by software, it is often indicated by an *error code*.

1.2.1 Mistakes

Mistakes can be made in a number of different ways. For example:

1. A misunderstanding in communication, such as confusing metric with imperial measurements.
2. A misinterpretation or misreading of the specification by a software developer, such as swapping the order of parameters by mistake.
3. Assuming defaults: for example, in Java integers have a default value of 0, but in C++ there is no default value.

1.2.2 Software Faults

It is helpful to have a classification of the types of faults. The classification can be used for a number of purposes:

1. When analysing, designing, or coding software: as a checklist of faults to avoid.
2. When developing software tests: as a guide to likely faults.
3. When undergoing software process evaluation or improvement: as input data.

There are a number of different ways to categorise these software faults, but no single, accepted standard exists. The significance of faults can vary depending on the circumstances. One representative categorisation[5] identifies the following 10 fault types:

Algorithmic A unit of the software does not produce an output corresponding to the given input under the designated algorithm.

Syntax Source code is not in conformance with the programming language specification.

Computation and precision The calculated result using the chosen formula does not conform to the expected accuracy or precision.

Documentation Incomplete or incorrect documentation.

Stress or overload The system fails to operate correctly when the applied load exceeds the specified maximum for the system.

[5] S.L. Pfleeger and J.M. Atlee, *Software Engineering: Theory and Practice*, 4th ed. Pearson Higher Education, 2010.

Capacity and boundary The system fails to operate correctly when data stores are filled beyond their capacity.

Timing or coordination The timing or coordination requirements between interacting, concurrent processes are not met. These faults are a significant problem in real-time systems.[6] where processes have strict timing requirements and may have to be executed in a carefully defined sequence.

Throughput or performance The developed system does not meet its specified throughput or other performance requirements.

Recovery The system does not recover to the expected performance even after a fault is detected and corrected.

Standards and procedure A team member does not follow the standards deployed by the organisation, which will lead to the problem of other members having to understand the logic employed or to find the data description needed for solving a problem.

It is very difficult to find industry figures relating to software faults, but one example is a study by Hewlett Packard[7] on the frequency of occurrence of various fault types, which found that 50% of the faults analysed were either *algorithmic* or *computation and precision* errors.

1.2.3 Software Failures

Classifying the severity of failures that result from particular faults is more difficult because of their subjective nature, particularly with failures that do not result in a program crash. One user may regard a particular failure as being very serious, while another may not feel as strongly about it. Table 1.2 shows an example of how failures can be classified by their severity.

Table 1.2 Sample classification of software failures.

Severity level	Behaviour
1 (most severe)	Failure causes a system crash and the recovery time is extensive; or failure causes a loss of function and data and there is no workaround.
2	Failure causes a loss of function or data but there is a manual workaround to temporarily accomplish the tasks.
3	Failure causes a partial loss of function or data where the user can accomplish most of the tasks with a small amount of workaround.
4 (least severe)	Failure causes cosmetic and minor inconveniences where all the user tasks can still be accomplished.

Hardware failures show a typical *bathtub* curve, where there is a high failure rate initially, followed by a period of relatively low failures, but eventually the failure rate rises again. The early failures are caused by manufacturing issues, and handling

[6] A real-time system is one with well-defined time constraints.
[7] Pfleeger and Atlee, *Software Engineering*.

and installation errors. As these are ironed out, during the main operational life of a product, the failure rate stays low. Eventually, however, hardware ages and wears out, and the failure rates rise again. Most consumer products follow this curve.

Software failures demonstrate a similar pattern, but for different reasons as software does not physically wear out. A typical curve for the failure rate of software is shown in Figure 1.2.

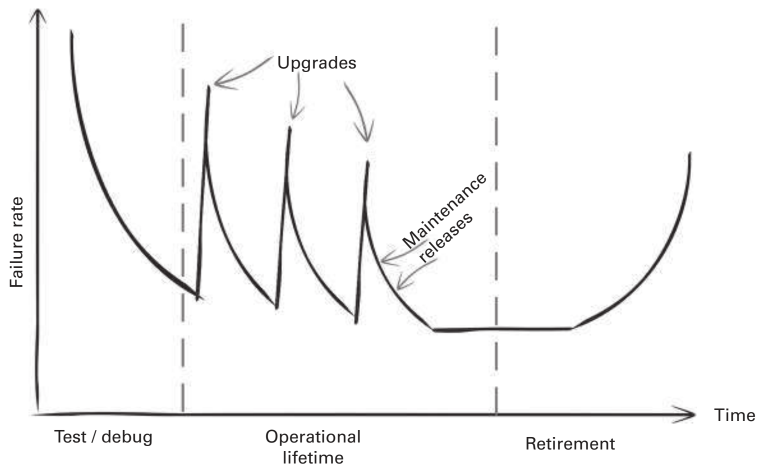

Figure 1.2 Failure rate for a software product over its life cycle.

During initial development, the failure rate is likely to fall rapidly during the test and debug cycle. After the first version is released, during the operational lifetime of the software, there may be periodic upgrades, which tend to introduce new failures, exposing latent faults in the software or introducing new faults. These upgrades may, for example, include additional features, changes required for integration with other software, or modifications required to support changes in the execution environment (such as operating system updates). Subsequent maintenance activity progressively lowers the failure rate again, reflecting an overall increase in code quality. Finally, the software is retired when it is no longer actively developed; this is particularly relevant to open-source software, where the original developer may stop maintaining their contribution. Eventually changes to the environment, or to software dependencies, may lead to failure again.[8]

The bathtub model is relevant to modern, Agile development projects with continuous integration of software features. New features are added on a regular basis, and software is frequently redesigned (referred to as refactoring). The changes introduced by this rapid rate of upgrading are likely to lead to new faults being introduced. Most

[8] For example, Python 3 was not fully compatible with Python 2, and Python 2 libraries that were not updated stopped working with libraries that were updated.

software is eventually replaced, or ceases to be maintained as a supported version, leaving existing users with an eventual rise in the failure rate.[9]

1.2.4 Need for Testing

There are a number of reasons why software has faults, or is perceived to have faults. For example, it is difficult to:

- collect the user requirements correctly;
- specify the required software behaviour correctly;
- design software correctly;
- implement software correctly; and
- modify software correctly.

There are two engineering approaches to developing correct systems: one is *forward engineering*, the other is based on *feedback*.

The ideal software development project, as shown in Figure 1.3, starts with the user and ends with a correct implementation. The development is completely reliable: each activity creates the correct outputs based on its inputs (specification) from the previous activity. The end product thereby matches its specification and meets the user's needs.

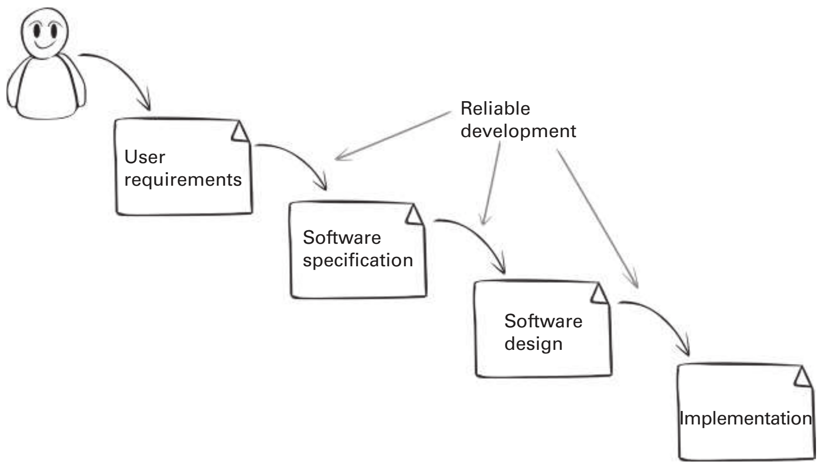

Figure 1.3 Ideal project progression using forward engineering.

In practice, however, all the development steps are subject to mistakes and ambiguities, leading to less-than-ideal results. To resolve this, each of the steps must be subject to a check to ensure that it has been carried out properly, and to provide an opportunity to fix any mistakes before proceeding. The verification and fixing following unreliable development at each step is shown in Figure 1.4.

[9] For example, support for the Windows 7 operating system ended at the beginning of 2020, leaving existing users with no security upgrades. A progressive rise in the failure rate of these systems can be expected until they are replaced with Windows 10.

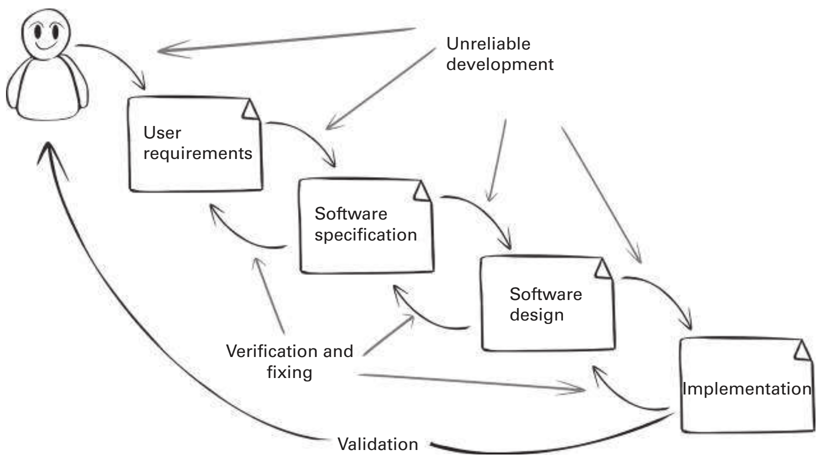

Figure 1.4 Realistic project progression with verification and validation.

For software products, in addition to checking that each individual step has been done correctly, it has been found in practice that a second form of checking is necessary: making sure that the implementation meets users' needs. The first form is referred to as verification, the second as validation.

1.3 The Role of Specifications

A *software specification* identifies the precise and detailed behaviour that the software is required to provide. It gives a *model* of what the software is supposed to do. These specifications play a key role in testing. In order to be tested, the correct behaviour of software must be known. This implies the need for detailed specifications (or software requirements[10]).

To support thorough testing of the code, specifications must describe both *normal* and *error* behaviour. Normal behaviour is the expected behaviour (or output) of the software when the inputs are not in error. Error behaviour is the expected results of the software when one or more inputs are in error, or will cause an error in processing.

Attempts have been made to develop test approaches based on *reasonable* behaviour of software – often deduced from the name of the method. This fails as not every developer and tester will have exactly the same expectations of reasonable behaviour, especially for error cases. Some examples include the following:

- If an invalid input is provided, should the method ignore it, return an invalid value, raise an exception, or write to an error log?
- If a temperature is calculated, should it be returned in degrees Celsius (°C), kelvin (K), or degrees Fahrenheit (°F)?

[10] Note the difference between software requirements and user requirements. Software requirements state what the software must do. User requirements state what the user wants to be able to do.

- Is zero to be treated as a positive number, a negative number or neither?
- Can an angle have a value of 0°? Can a shape have an area of 0 cm^2? Or even a negative area?

In general terms, the expected behaviour of a program in handling errors in the input data is to indicate that this has occurred – for example by returning an error code, or raising an exception. But in order to test the software, the exact details of this must be defined. These specify exactly what an error in the input is, and how such errors should be notified to the caller. In summary, in order to test software properly, detailed specifications are necessary.[11]

Note that often a tester will need to convert a specification from a *written English* or natural-language form to one that is more concise and easier to create tests from.

1.4 Manual Test Example

Consider a program `check` for an online shop that determines whether a customer should get a price discount on their next purchase, based on their bonus points to date, and whether they are a gold customer.

Bonus points accumulate as customers make purchases. Gold customers get a discount once they have exceeded 80 bonus points. Other customers only get a discount once they have more than 120 bonus points. All customers always have at least one bonus point.

The program returns one of the following:

- `FULLPRICE`, which indicates that the customer should be charged the full price.
- `DISCOUNT`, which indicates that the customer should be given a discount.
- `ERROR`, which indicates an error in the inputs (`bonusPoints` is invalid if it is less than 1).

The inputs are the number of `bonusPoints` the customer has, and a flag indicating whether the customer is a gold customer or not (true or false, respectively). For the sake of simplicity, we will ignore illegal input values at the moment (where the first parameter is not a number, or the second parameter is not a Boolean).

Output from some demonstration runs of the program, using the command line,[12] are shown in Figure 1.5. All console output will be shown like this.

These results look correct: a regular customer with 100 points is charged the full price, a gold customer with 100 points is given a discount, and −10 is an invalid number of points.

So does this method work correctly? Does it work for every possible input? Next, we will consider the theory of software testing to try to answer these questions.

[11] One exception is in stability testing, where tests ensure that the software does not crash.
[12] See Section 14.4.2 for details of how to run the examples.