Chapter - 1

Introduction to Testing
CHAPTER – 1

INTRODUCTION TO TESTING

1.1 TESTING

We need to look at testing as a continuous activity throughout the entire development cycle, rather than a phase or step in the cycle. Testing should be the measurement of software quality not merely a defect finding process. The described methodology can be called verification and validation process rather than a test process because testing is only a section of the required mechanism that removes errors, increases software quality, and produces user centered systems.
Verification and validation (V & V) has been described by Boehm, as, 

- Verification: Are we building the product right?
- Validation: Are we building the right product?

Verification is the checking or testing of items, including software, for conformance and consistency with an associated specification. Software testing, is just one kind of verification. Validation is the process of checking that what has been specified is what the user actually wanted.

Many of the software based systems currently under development are highly interactive systems which usually have a graphical user interface (GUI) or deal with multimedia (Real-Time) data. The introduction of client-server, distributed systems, and now Internet applications require specialized V & V procedures.

The Key to simultaneously reducing development time while increasing software quality is to prevent errors at the beginning stages of development. The cost for fixing a bug is having almost an exponential relation with the phase of the project. Studies and analysis have shown that the cost of fixing a bug at the final stages is almost 10 times more that the cost for fixing it at the initial stages.

The later an error is detected in the lifecycle, the more expensive it is to correct. Perry states that this cost must be paid for four different times:

1. Firstly, in the development of the erroneous program.
2. Secondly, in the cost of testing the system to reveal the error.
3. Thirdly, in removing the error from the specification, and code.
4. And finally fourthly, in retesting the system to prove that it is now correct.

Shooman describes the reasons for increases in error removal costs a project progresses as:

1. Testing becomes more complex and costly.
2. Documentation of changes becomes more widespread and costly.
3. Communication of problems and changes involved many people.
4. Repeating of previous tests (regression testing) becomes costly.
5. Once operation is begun, the development team is disbanded and reassigned.

1.1 V- Model in the software life-cycle

In software engineering a software-cycle and be illustrated as a V-Model diagram.
As shown in figure 1.2, there are different stages in the software lifecycle ranging from the “Requirements analysis” stage to the “Coding” stage. Each stage has corresponding test phase. The reliability is achieved through the different levels of testing, each providing a greater degree of confidence in the quality of the software.

1.2 COMMON MISTAKES DONE IN TESTING

It's easy to make mistakes when testing software or planning a testing effort. Some mistakes are made so often, so repeatedly, by so many different people, that they deserve the label Classic Mistake.

Classic mistakes cluster usefully into four groups,
- The Role of Testing: who does the testing team serve, and how does it do that?

A first major mistake people make is thinking that the testing team is responsible for assuring quality. This role, often assigned to the first testing team in an organization, makes it the last defense, the barrier between the development team (accused of producing bad quality) and the customer (who must be protected from them). It's characterized by a testing team (often called the “Quality Assurance Group”) that has formal authority to prevent shipment of the product. That in itself is a disheartening task: the testing team can’t improve quality, only enforce a minimal level. Worse, that authority is usually more apparent than real. Discovering that, together with the perverse incentives of telling developers that quality is someone else’s job, leads to testing teams and testers who are disillusioned, cynical, and view themselves as victims. We’ve learned from Deming and others that products are better and cheaper to produce when everyone, at every stage in development, is responsible for the quality of their work.

- Planning the Testing Effort: how should the whole team’s work be organized?

It's not unusual to see test plans biased toward functional testing. In functional testing, particular features are tested in isolation. In a word processor, all the options for printing would be applied, one after the other. Editing options would later get their own set of tests.
But there are often interactions between features, and functional testing tends to miss them. For example, you might never notice that the sequence of operations open a document, edit the document, print the whole document, edit one page, print that page doesn’t work. But customers surely will, because they don’t use products functionally. They have a task orientation. To find the bugs that customers see - that are important to customers - you need to write tests that cross functional areas by mimicking typical user tasks. This type of testing is called scenario testing, task-based testing, or use-case testing.

- Personnel Issues: who should test?

The classic mistake is recruiting testers from the ranks of failed programmers. There are plenty of good testers who are not good programmers, but a bad programmer likely has some work habits that will make him a bad tester, too. For example, someone who makes lots of bugs because he’s inattentive to detail will miss lots of bugs for the same reason. So how should the testing team be staffed? If you’re willing to be part of the training department, go ahead and accept new programmer hires. Accept as applicants programmers who you suspect are rejects (some fraction of them really have gotten tired of programming and want a change) but interview them as you would an outside hire. When interviewing, concentrate less on formal qualifications than on intelligence and the character of the candidate’s thought. A good tester has these qualities
• Methodical and systematic.
• Tactful and diplomatic (but firm when necessary).
• Skeptical, especially about assumptions, and wants to see concrete evidence.
• Able to notice and pursue odd details.
• Good written and verbal skills (for explaining bugs clearly and concisely).
• A knack for anticipating what others are likely to misunderstand. (This is useful both in finding bugs and writing bug reports.)
• A willingness to get one’s hands dirty, to experiment, to try something to see what happens.

• The Tester at Work: designing, writing, and maintaining individual tests.

When testing, Tester has to decide how to exercise the program, and then do it. The doing is ever so much more interesting than the deciding. A tester’s itch to start breaking the program is as strong as a programmer’s itch to start writing code - and it has the same effect: design work is skimmed, and quality suffers. **Paying more attention to running tests than to designing them** is a classic mistake. A tester who is not systematic, who does not spend time laying out the possibilities in advance, will overlook special cases. They may be the same subtle ones that the programmers overlooked.
Concentration on execution also results in unreviewed test designs. Just like programmers, testers can benefit from a second pair of eyes. Reviews of test designs needn’t be as elaborate as product design reviews, but a short check of the testing approach and the resulting tests can find significant omissions at low cost.

1.3 TYPES OF TESTING

Testing Techniques execute the code. There are many different types of testing. The following are the most common testing types:

- Unit Testing
- Integration Testing
- Regression Testing
- System Testing
- Beta Testing
- Acceptance Testing

1.3.1 Unit Testing

Unit testing is the lowest level of testing. It is done by developers, who make sure that their code works as it should. Units that are tested can be single functions, modules, or combination of modules. Unit tests are done in isolation. Most common way to do unit testing is to automate the tests. Automated unit tests are tests that run a piece of code, check the result, compare it to expected and report the status.
of the tests. Everything is done automatically. Defects found using unit testing are usually easy to fix, and when developers fix their own defects, the rest of the software is not affected by these defects. Usually unit testing finds defects in the external quality characteristics, but it can also find defects in the internal quality characteristics, especially in testability. Unit testing is not free, because it requires that each developer writes test cases of their code, and this will take some time, depending on how many tests are being written. Depending on the framework used, unit testing can be very easy to use, but if there is no framework available, it could take too much effort to write and execute unit tests. There is a lot of other benefits unit testing. Those benefits will be listed in the next chapter.

1.3.2 Integration Testing

Integration testing includes testing of combined parts of application to determine if they work correctly together. Usually integration testing is done after unit testing to check that different units work together, but it can also be applied to bigger components that single units. Usually integration testing finds defects in external quality characteristics, but sometimes it can find defects also in the internal quality characteristics also. Integration testing is not very expensive, because it is usually done by one or two people. Integration testing is not as easy to use as unit testing, but it is not very difficult either. Integration testing does not have a lot of other benefits. Knowledge of other modules in the software rises when integration testing is performed.
1.3.3 Regression testing

Regression testing is not a testing level. It is a testing technique that can be applied to any level of testing. Regression testing means that same tests are executed again to make sure that change to the software did not have any unwanted side effects on other parts of the software. Regression testing finds defects in the external quality characteristics. It does not detect defects in the internal quality characteristics. If regression testing is automated, it is very inexpensive, if not, it required normal testing effort. Automated regression testing is also very easy to use, but manual regression testing takes a lot more effort. Regression testing does not have any additional benefits; it is just used to finds defects in the software.

1.3.4 System Testing

System testing is done after software has been created. I is what most people understand by testing. It is the process of executing the software with the intent of finding errors. This includes finding errors in all of software’s external quality characteristics. System testing has a particular purpose to compare the system to its original specifications. Without the original documentation system testing cannot be completed. System testing is usually done in the end of the project, which means that every defect found during system testing often required fixing the software. This leads to the problem where the corrections done during the system testing might break something else in the software and this might not get notices, because the software is already on its way out. System testing usually determines the quality of the software.
instead of improving it. System testing is not inexpensive, because it usually requires a lot of manual work. If system testing is automated, it will be more inexpensive. However, automation in the first place is expensive. Professional system testing is not very easy, because it requires proper documentation and execution of test plans. People, who execute the tests, should also be trained. When system testing is done by people, it adds the knowledge of the software; otherwise it does not have many other benefits.

1.3.5 Beta Testing

Beta Testing is testing usually done by external testers, often by customers in their own environment. Beta Testing is used to make sure that the software works correctly in customer’s environment. Extensive beta testing can be very useful in detecting defects, but if it is not done properly, it might not improve the quality of the software at all. Beta Testing is usually very expensive, but if the actual beta testers are not paid, it is not that expensive. Beta Testing is not very easy to use, because it required a lot of communication between the organization an the testers. Sometimes it might even require too much work to start beta testing. Beta Testing can be a valuable tool, not only in finding defects in software, but also in getting new feature ideas.

1.3.6 Acceptance Testing

Upon acceptance testing, the acceptance of the completed software is based. This will often use a subset of the system tests, witnesses buy the customers for the software or system.
1.3 WHAT TESTS TO WRITE?

The tests that are written should be isolated and automatic. Each test doesn’t interact with the others are written. This way we avoid the problem that one test fails and causes testing 21 hundred other failures. Tests should also be automatic. Tests are most valuable when the stress level rises, when people are working too much, when human judgment starts to fail. It is impossible to test absolutely everything without the tests being as complicated and error-prone as the code. We should write tests for things that might break. If code is so simple that it can’t possible break in practice, we shouldn’t write tests for it. If possible we would only write those tests that pay off. Since beforehand it is not possible to know which tests will pay off, tests are written that might pay off. As we test, we reflect on which kinds of tests tend to pay off and which don’t, and you write more of the ones that do pay off, and fewer of the ones that don’t.

1.4 AUTOMATED TESTING

So far a number of types of tests have been discussed. The goal of this thesis is to evaluate the effect the Test Feedback chain has on the overall software quality. This type of testing that we use in this chain is called automated testing. We will introduce three types of testing that we will automate for this case study, namely:

1. Unit Testing
2. Functional Testing
3. Performance Testing
4. Profiling

These forms of testing all have their own purpose and own way of implementation.

In the testing world, there are a lot names that actually mean the same thing. Integration testing, System testing and Acceptance testing are all functional tests. They test if the functionality of the part they are testing is meeting the external requirements.

Using automation, tests can be run faster, in a consistent manner and with fewer over-heads. It can kill the problems of test scheduling, the costs of testing and defect reporting. Automation is the replacement of supplementation of manual testing with a suite of test programs. Benefits include increased software quality, reduced time to market, reusable test procedures, and reduced testing costs.

Automated unit and functional testing falls under the regression testing category; Regression testing is testing that a program has not regressed, i.e., that the functionality that was working yesterday is still working today.

1.5.1 Unit testing

Units are the smallest building blocks of software. This unit test is the lowest level of testing performed during software development, where individual units of software are tested in isolation from other parts of a program.
Definition 3 Unit Testing:

Unit tests are small programs that run in batches and test classes. Each test typically sends a class a fixed message and verifies it returns to a known answer.

If it is known beforehand that $A + B$ should be 4, an assertion can be done on that expression:

```
ASSERT (A + B = 4);
```

This is called a unit test.

The most common method of testing a unit is the White Box Testing technique. This method requires the tester to gain some knowledge about the internal structure of the component, since some of the tests should be designed to address the internals of the component based on its structure.

This way, it is possible to verify that what is inspected is the same as what is expected. The tests will tell if what has been implemented is what is thought to be implemented.

This Philosophy behind unit testing is: create the test (the expectation) first before coding starts. This philosophy is adapted from Extreme Programming. When you create your tests first, before you code, you will find it much easier and faster to create your code. This methodology requires that we write unit tests for every
function we add, and that we maintain those tests. We can’t integrate any code with failing unit tests. As the code base grows, these tests allow developers to integrate changes with confidence. If unit tests are run whenever changes to the code are made, you’ll find out immediately if your changes broke something.

There is also a benefit to system design. It is often very difficult to unit test some software systems. These systems are typically coded first and tested second, often by a different team entirely. By creating tests first the design will be influenced by a desire to test everything of value to the customer. The designs will reflect this by being easier to test.

If you want to add tests later in the development cycle, it should be made simple and easy for the programmers to add new tests. If adding a unit test can be done quickly and easily, the combined time it takes to create a unit test and create some code to make it pass, is about the same as just coding it up straight away.

1.5.2 Functional Testing

Definition 4 Functional testing:

Functional testing, or integration testing, is testing concerned with the entire system, not just small pieces (or units) of code.

Certain scenarios are thought up in front and we then try to simulate those scenarios. Functional test capture user requirements in a useful way. Traditional
development captures requirements in use cases. Usually, people argue about the use cases and spend a lot of time refining them. When they're finished, all they have is paper. Functional tests are like self validating use cases. Whereas unit tests are written from a programmer's perspective, functional tests are written from a user's perspective.

Functional testing activities assess whether a software product meets the requirements of a particular specification or standard. They are used to evaluate whether the software product implements each of the specific requirements of the standard or specification.

1.5.3 Performance testing

Performance testing measures how well the software system executes according to its required response times, throughput, CPU usage, and other quantifies features in operation by comparing the output of the software being tested to predetermined corresponding target and reference values.

<table>
<thead>
<tr>
<th>Definition 5 Performance testing:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance testing is testing conducted to evaluate the compliance of a system or component with specified performance requirements. Often this is performed using an automated test tool to simulate large number of users.</td>
</tr>
</tbody>
</table>
“Performance testing is also commonly known by the other names and/or associated with other testing activities, such as stress testing, capacity testing, load testing, volume testing, and benchmark testing. These various performance testing activities all have approximately the same goal: measuring the software product under a real or simulated load”

(Beizer)

Typically performance testing cannot be performed earlier in the life cycle because a fully or nearly fully developed software product is needed. In fact, proper performance testing may require that the software product be fully installed in a real or simulated operational environment.

1.5.4 Profiling

Definition 6 Profiling:
Profiling is any analysis method that can be used to examine the runtime behavior of your programs. By using profiling information, it is possible determine how much time a piece of code is called relative to other code, and thus determine which sections of your code are working efficiently.

By profiling code we get a formal summary or analysis of data on that code. This means that we record and process information that to be profiled code. The type of profiling we are using is called: Function Profiling. This result in the profiler producing information that includes numbers like function hits and the time spent in
functions. By knowing how frequently a function is called and how many times is spent in that function, you can more accurately gauge the importance of optimizing that piece of code. Function profiling is good for detecting inefficient code.

Profiling does not fall under the category of testing because we are not actually testing something, meaning we are not looking if some properties of the to be profile functions are valid, but it is more an analyzing tool. As such, this section actually does not belong under the section Automated testing, but it does fall under the category techniques used for our case study in order to get valuable information to analyze code. The profiler however, should be used to make programs run better, not to find bugs.

### 1.6 RELATIVE EFFECTIVENESS OF QUALITY TECHNIQUES

Defect removal efficiency, the percentage of defects found using a quality improvement technique, can be used to compare different techniques. Defect removal efficiency is calculated by counting the number of defects found before releasing the product and dividing it by the total number of defects found in the product after it has been available to the customers for some time. In the table below effectiveness of different quality improvement techniques is shown. This table is presented in literature in. As can be seen from the modal rates in the table, no single technique can discover more than 75 per cent of the defects. When system testing is the only used technique, only 40 per cent of the defects are found. It is clear from this table that organizations should use many different techniques to eliminate as many defects as possible.
<table>
<thead>
<tr>
<th>Removal Step</th>
<th>Lowest Rate</th>
<th>Modal Rate</th>
<th>Highest Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Informal design reviews</td>
<td>25%</td>
<td>35%</td>
<td>40%</td>
</tr>
<tr>
<td>Formal design inspections</td>
<td>45%</td>
<td>55%</td>
<td>65%</td>
</tr>
<tr>
<td>Informal code reviews</td>
<td>20%</td>
<td>25%</td>
<td>35%</td>
</tr>
<tr>
<td>Formal code inspections</td>
<td>45%</td>
<td>60%</td>
<td>70%</td>
</tr>
<tr>
<td>Modeling or prototyping</td>
<td>35%</td>
<td>65%</td>
<td>80%</td>
</tr>
<tr>
<td>Personal desk checking of code</td>
<td>20%</td>
<td>40%</td>
<td>60%</td>
</tr>
<tr>
<td>Unit testing</td>
<td>15%</td>
<td>30%</td>
<td>50%</td>
</tr>
<tr>
<td>Integration testing</td>
<td>25%</td>
<td>35%</td>
<td>40%</td>
</tr>
<tr>
<td>Regression testing</td>
<td>15%</td>
<td>25%</td>
<td>30%</td>
</tr>
<tr>
<td>System testing</td>
<td>25%</td>
<td>40%</td>
<td>55%</td>
</tr>
<tr>
<td>Low volume beta testing (&lt;10 sites)</td>
<td>25%</td>
<td>35%</td>
<td>40%</td>
</tr>
<tr>
<td>High volume beta testing (&gt;1000 sites)</td>
<td>60%</td>
<td>75%</td>
<td>85%</td>
</tr>
</tbody>
</table>

Although test automation has many advantages, it can also have some drawbacks. If not planned carefully, it may lead to a poor quality of testing that in turn lead to poor quality product. Next chapter will deal with the problem with automation and the frame worked solution approach.

1.7 SURVEY OF TESTING TECHNIQUES
A survey of the application of software testing techniques used throughout 80 organizations resulted in the following table:

Table 1.4: Survey outcome

<table>
<thead>
<tr>
<th>Percentage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>60%</td>
<td>Indicated that their companies had implemented some form of testing standard. Some of them admitted that the standards were out of date, or not followed.</td>
</tr>
<tr>
<td>65%</td>
<td>Acknowledged that their testing date and not quality driven, so that when the allocated development time is exceeded the software is released, and all further testing activity is suspended whether the product is completely validated or not</td>
</tr>
<tr>
<td>21%</td>
<td>Maintained reused test information.</td>
</tr>
<tr>
<td>17%</td>
<td>Stated that tests were never reused.</td>
</tr>
</tbody>
</table>

The survey results show that a lot of companies are involved in testing in some kind of way, but struggle with its implementation and with adapting to an effective testing culture.
1.8 BUSINESS PROBLEM STATEMENT

At the onset of the Finance Systems Roadmap program, the capability assessment of the current state General Ledger system functionality and technical architecture review identified the need for:

1. Better system integration
2. Better support for GAAP & STAT schedules through traceability to source system data
3. Improvement of data quality up-stream
4. Better Integration with operational warehouse
5. Dimensions required beyond the vanilla chart fields available in PeopleSoft GL
6. Automation of current state manual paper driven processes

Regulatory reporting requirements for the insurance industry require significantly more dimensions for financial information when compared with a typical GL system. The current state GL is highly customized given it contains a large number of dimensions and values to support regulatory reporting. Current state financial systems often do not retain the adequate level of transactional detail to support balancing and reconciliation efforts to the source of the financial data.
With these needs in mind, the vision of the overall program is to build an integrated platform that will be the single source of the truth for management information needs. This vision entails utilization of a common financial language (CFL) encompassing all the components of the future state common platform. This repository of common accounting rules and transformation logic coupled with a single integrated accounting engine employing a hub and spoke architecture (middleware platform to which applications are integrated) is the envisioned, future state solution for CFL. The Accounting Center is the proposed system in the future state common platform enabling CFL, better support for schedules, better traceability to source systems, simplified reconciliation and a more streamlined close processes.