Emerging trends with legacy techniques in testing software product

#1 Chandrashekhar P K, #2 Nandish Ramegowda, #3 Sandeep A P
#1 Software Engineer, Cisco Systems, Bangalore
#2 QA Lead Engineer, Cisco Systems, Bangalore
#3 Network Consulting Engineer, Cisco Systems, Bangalore
Abstract

The importance of testing the software product has become inevitable part of the large scale organizations today. There is a potential need for cost effective testing and quality assured software. The testing techniques have to be improved and fastened to increase the productivity and success for the business. The majority of manual testing need to be automated to provide more bandwidth for the QA engineer to concentrate on high intellect task and avoid time consuming and redundant tasks with automation. The main target of automation is regression, as it has to be done every major or minor release of the software product. Also there is a need for improved test case writing techniques and tools that helps the engineer to fasten the testing activities. This paper mainly concentrates on the regression techniques, test strategies and tools of testing.

1. Software Testing

Software testing is the process of evaluation a software item to detect differences between given input and expected output. Also to assess the feature of A software item. Testing assesses the quality of the product. Software testing is a process that should be done during the development process as shown Fig 1. In other words software testing is a verification and validation process.

Verification is the process to make sure the product satisfies the conditions imposed at the start of the development phase. In other words, to make sure the product behaves the way we want it to. Validation is the process to make sure the product satisfies the specified requirements at the end of the development phase. In other words, to make sure the product is built as per customer requirements.

2. Basics of software testing

There are two basics of software testing: black box testing and white box testing. Black box testing is a testing technique that ignores the internal mechanism of the system and focuses on the output generated against any input and execution of the system. It is also called functional testing. White box testing is a testing technique that takes into account the internal mechanism of a system. It is also called structural testing and glass box testing. Black box testing is often used for validation and white box testing is often used for verification.

Types of testing

There are many types of testing like [1]

- Unit Testing
- Integration Testing
- Functional Testing
- System Testing
- Stress Testing
- Performance Testing
- Usability Testing
- Acceptance Testing
- Regression Testing
- Beta Testing

According to Pierre Audoin Consultants [2], companies around the world invest more than $50 billion per year on applications testing and quality assurance. Research firms such as IDC and Forrester report a five-year compound annual growth rate (CAGR) of 15.4 percent with spending reaching nearly $19.3 billion by 2015 on testing service alone.
Regression testing is an important and expensive activity that is undertaken every time a program is modified to ensure that the modifications do not introduce new bugs into previously validated code as shown in Fig 2. An important research problem, in this context, is the selection of a relevant subset of test cases from the initial test suite that would minimize both the regression testing time and effort without sacrificing the thoroughness of regression testing. Researchers have proposed a number of regression test selection techniques for different programming paradigms such as procedural, object-oriented, component-based, database, aspect, and web applications. In this paper, we review the important regression test selection techniques proposed for various categories of programs and identify the emerging trends.

Software maintenance activities, on an average, account for as much as two-thirds of the overall software life cycle costs. Maintenance of a software product is frequently necessitated to fix defects, to add, enhance or adapt existing functionalities, or to port it to different environments. Whenever an application program is modified for carrying out any maintenance activity, resolution test cases are designed and executed to check that the modified parts of the code work properly. Regression testing (also referred to as program revalidation) is carried out to ensure
that no new errors (called regression errors) have been introduced into previously validated code (i.e., the unmodified parts of the program) [3]. Although regression testing is usually associated with system testing after a code change, regression testing can be carried out at either unit, integration or system testing levels. The sequence of activities that take place during the maintenance phase after the release of a software. The figure shows that after a software is released, the failure reports and the change requests for the software are compiled, and the software is modified to make necessary changes. Resolution tests are carried out to verify the directly modified parts of the code, while regression test cases are carried out to test the unchanged parts of the code that may be affected by the code change. After the testing is complete, the new version of the software is released, which then undergoes a similar cycle. Regression testing is acknowledged to be an expensive activity. It consumes large amounts of time as well as effort, and often accounts for almost half of the software maintenance costs. The extents to which time and effort are being spent on regression testing are exemplified by a study that reports that it took 1000 machine-hours to execute approximately 30,000 functional test cases for a software product. It is also important to note that hundreds of man-hours are spent by test engineers to oversee the regression testing process; that is to set up test runs, monitor test execution, analyze results, and maintain testing resources, etc. Minimization of regression test effort is, therefore, an issue of considerable practical importance, and has the potential to substantially reduce software maintenance costs.

Regression test selection (RTS) techniques select a subset of valid test cases from an initial test suite (T) to test that the affected but unmodified parts of a program continue to work correctly. Use of an effective regression test selection technique can help to reduce the testing costs in environments in which a program undergoes frequent modifications. Regression test selection essentially consists of two major activities: identification of the unmodified parts of the program that are affected by the modifications. Test case selection - This involves identification of a subset of test cases from the initial test suite T which can effectively test the unmodified parts of the program. The aim is to be able to select the subset of test cases from the initial test suite that has the potential to detect errors induced on account of the changes.

complex systems, it is usually extremely difficult to manually identify test cases that are relevant to a change. This approach often leads to a large number of test cases being selected and rerun even for small changes to the original program, leading to unnecessarily high regression testing costs. What is probably more disconcerting is the fact that many test cases which could have potentially detected regression errors could be overlooked during manual selection. Another problem that surfaces during regression testing stems from the fact that testers (either from the same organization or from third-party companies) are usually supplied with only the functional description of the software, and therefore lack adequate knowledge about the code to precisely select only those test cases that are relevant to a modification.

A large number of RTS techniques have been reported for procedural and object-oriented programs, each aimed at leveraging certain optimization options. These techniques trade-off differently with regards to the cost of selection and execution of test cases and fault-detection effectiveness. In the recent past, the problem of RTS has actively been investigated and new approaches have emerged to keep pace with the newer programming paradigms. During the last decade, there has been a proliferation in the use of different programming paradigms such as component-based development, aspect-oriented programming, embedded and web applications, etc. It is, therefore, not surprising that a number of RTS techniques have been proposed for component-based aspect programs [3], web applications etc.

RTS techniques have been reviewed by several authors. In Rothermel and Harrold have proposed a set of metrics to evaluate the effectiveness of different RTS techniques. Baradhi and Mansour, Bible et al. and Graves et al. have performed experimental studies on the performance and effectiveness of different RTS techniques proposed for procedural programs. Based on these studies, it is difficult to choose any technique as the best because these empirical studies have been performed on different categories of programs and also under different conditions. This lead Engström et al. to perform a qualitative study of Rothermel and Harrold have formally defined the regression test selection problem as follows: Let X be an application program and X′ be a modified version of X. Let T be the test suite developed initially for testing X. An RTS technique aims to select a subset of test cases T′ ⊆ T to be executed on X′, such that every error detected when X′ is executed with T is also detected when X′ is executed with T.

Leung and White have observed that the use of an RTS technique can reduce the cost of regression testing compared to the retest-all approach, which involves running the entire test suite T to revalidate a modified program P, only if the cost of selecting a reduced subset of test cases to be run on P′ is less than the cost of running the tests that the RTS technique omits. The retest-all approach is considered impractical on account of cost, resource and delivery schedule constraints that projects are frequently subjected to. Another approach is to randomly select test cases from T to carry out regression testing. However, random selection of test cases may fail to expose many
regression errors. RTS techniques aim to overcome the drawbacks associated with the retest-all approach and in random selection of test cases by precisely selecting only those test cases that test the unmodified but affected parts of the program.

Though substantial research results on RTS have been reported in the literature, several studies [35, 36] show that very few software industries deploy systematic test selection strategies or automation support during regression testing. The approaches that are most often used in the industry for identification of relevant regression test cases are either based on expert judgment, or based on some form of manual program analysis. However, selection of test cases based on expert judgment tends to become ineffective and unreliable for large software products. Below are the few things need to be considered while creating regression suit.

1. At what levels should you perform your regression tests?
   As you know, software testing happens at a number of levels (e.g. unit testing, integration testing, system testing and so on) [4]. Since the focus of testing is different at different levels, you are going to have a better likelihood of finding regression defects if you perform regression testing at different levels. For example, you may find defects in unit testing which would not have been captured by your existing system tests. You may find defects in integration testing which would not have been captured by any unit tests.

2. Which test cases do you execute (the existing test cases or new test cases or a combination)?
   First of all, you should be aware of the impact of the changes to the software (since the last regression test). Then select a suitable set of test cases that sufficiently exercise the impacted areas of the software. You may want to use the existing test cases if they are sufficient in their coverage. Further, they should have been updated along with the prior changes to the software e.g. the test cases are updated based on the defects discovered in the past. If you know that the existing test cases have skimpy coverage or out of date, you may want to create new test cases for your regression test. You should also ensure that there are no duplicate tests in your chosen regression test suite.

3. How frequently should you run the regression test?
   This depends on the cost you incur in running your regression test and the value you receive out of it. Even if you don't run a regression test every day (note that some teams do), you should at least run one complete regression test before the software is released to production or to your client.

4. Which part of the regression test should be automated and which part should be manually executed?
   Tests which are stable, repeated frequently, simple and require no intervention by the tester are good candidates for automation.

5. How do you ensure that your regression tests are effective?
   Your regression tests should be able to discover defects. Upstream regression tests should discover a greater number or the more severe defects or at least discover defects more easily than downstream tests.

6. How do you ensure that your regression tests are optimized?
   If you are aware of the scope and timing of the build process, you can align your regression tests with it. This will lead to an optimal number of regression test runs[4].
   Further, you should examine your regression test cases to eliminate duplicate test cases, merge test cases wherever possible and automate tests (based on the criteria above) to minimize the time/effort it takes to run your regression test.

3. Ways of testing software

3.1 Manual Testing

For most customers, the majority of functional testing today is performed manually as shown in Fig 3. This is primarily due to the fact that deep subject matter expertise is needed to understand the business process variations and rules, and the subject matter experts also have other responsibilities besides testing. Therefore, they are called in as needed to verify the functional needs for their area [5].

There are several drawbacks to a manual approach, the most obvious being the amount of time it takes. As described above, the same process may be executed hundreds of times for each transport, and this not only extends the project timeline but it consumes scarce and valuable resources from the business. If the project is running late test coverage must be sacrificed, which in turn increases the risk of production failures or roll-backs which are
perhaps even more costly. There are other less obvious risks. One is that manual test approaches tend to be more informal than others. That is, the test cases and techniques are subject to the skills and preferences of the tester, leading to inconsistency and unreliability. This makes the coverage and resulting quality unpredictable and not repeatable from one transport to the next.

Another downside is that manual testing is so time-consuming that testers may not have the time to thoroughly document the tests or the results, or they do so in an inconsistent way. Even if test documentation was originally created for the initial roll-out, it may not be kept current for changes. This leads to a lack of management visibility and an inability to support compliance audits or regulatory requirements.

Finally, it is difficult to coordinate the testing of end to end business processes that span various solution modules. Business process experts are often organized around functional areas, yet the risk of up and downstream impact from changes dictates that processes be tested across departments and modules. Because of these challenges, many companies have sought to automate their functional testing of their solutions using tools commonly known as record/playback.

![Fig 3. Manual Testing Activities](image)

### 3.2 Automation

**Record/Play**

Record/play is a seductive idea: simply perform the test manually and record the steps into a script that can be replayed multiple times. Unfortunately, while this approach is presented as easy to implement, it produces poorly structured, undocumented and unstable tests that are not reusable or maintainable.

Recorded scripts are unstable because they are sensitive to the slightest changes. If the application is running more slowly at some times than others, the script can get out of synchronization and result in errors. Or if an unexpected condition arises, the script has no logic to recover and continue. Even changes in lists of data can cause recorded scripts to fail. These types of scripts also create a high maintenance overhead because they contain hard-coded data. This means that if you want to test a hundred different order variations, for example, your script will repeat the same steps hundreds of times. Naturally if a change is made to the order process, the script will have to be updated hundreds of times. The fig 4 dictates the automation life cycle.

The lack of any logic within these scripts also precludes making decisions or changing the workflow based on test results. For example, a particular material code may trigger a special window to appear that otherwise does not. Using the record/play approach requires that there be two separate scripts—one for each condition. When you take into account the large number of similar cases, this leads to a high degree of redundancy. The only way to overcome these challenges with recorded scripts is to employ the underlying scripting language to manage timing, detect and recover from errors, make decisions, parameterize the data values and read external data sources. Scripting is
essentially programming and requires advanced technical skills, which excludes the very business process experts whose knowledge is required for effective testing.

Companies who decide to make the investment in coding soon find themselves with thousands of lines of code that had to be maintained. Often the code was written by contractors who are no longer available and new coders prefer to rewrite the code. This code itself is rarely documented and creates an unreasonable maintenance burden, reducing the time and resources available to add new tests.

As a result, most test automation efforts are abandoned. It simply takes too long to develop and maintain all of the code, and costs too much to have skilled coders on staff or as consultants, that most revert to manual testing after much frustration and expense. This is evidenced by literally billions of dollars of test tool shelfware that is abundant in all market segments.

4. Code Coverage Myth

People often ask this question: I have 90% code coverage on my code how am I stilling seeing bugs being reported? Well..how about if I say that even if you do 100% code coverage, the testing won’t be foolproof. It is a myth that by increasing code coverage a better software can be written. The problem lies with the concept, with notion that touching or exercising lines of code with a test make sure that bugs no longer exists.

In fact code coverage is only a numerical as shown in fig 5 which is good to have but should not be relied upon [6]. There is no substitute in world of testing for good test cases and only good test cases can make sure that your code is tested fully, that you have explore all possibilities of finding bugs. The way you design your test cases speaks volume about your testing rather than your code coverage results. So next time you come across dilemma of increasing code coverage or writing good test cases, I would suggest you to prefer later. If you write good test cases chances are you will get good coverage but reverse is absolutely not certain at all.

Let us follow it by a simple example to substantiate what I have just said. Here you go: $x = \frac{Y}{Z}$. Now you can fully execute this line of code with any value of $Y$ and $Z$ (say $Y = 1$ and $Z = 1$) and get 100% code coverage. But to see that this code is missing an exception handling (which is a BIG FAT BUG) you need to come up with a test case with $Y = 1$ and $Z = 0$. It also confirms what I stated earlier that a better designed test case can give you 100% code coverage but reverse is not true. So I would encourage you to write better test cases then going after code coverage number.
5. Importance of Negative Test Cases

Negative test cases are written to verify that software deals appropriately with the situation for which it is not programmed. For example if I write a piece of code for Addition Application, with specs written as “Application takes two inputs to give an output which is summation of input” it should show some kind of error if division or any other arithmetic operation is attempted on input. It should neither crash nor remain silent but should throw a reasonably worded error message to user. In other words, negative test cases are written contrary to what is mentioned in the functional spec to break the functionality. The idea is to check if the application is stable and the program is doing what it is intended to do.

Negative test cases, therefore have a pivotal role in testing a product. Care must be taken while writing negative test cases and following should be considered:

1. **Negative test cases coded as positive**: This happens when the functional specs is not precise and allows room for making assumptions. In this scenario the tester is basically testing for presence of a bug but unable to detect it as a bug. The test which should fail is actually passing and the tester has no clue that the passing test is indicating a source of bug. Take an example of our simple Addition Application discussed above. Now if I have a test case 2+2 and 2*2 (remember multiplication is recurring addition) and I am just verifying the result (which should be 4) then I would not able to see that the second case is a bug and should be detected by this negative test case. If I had written the specification as: “Application take two input and + operator to give an output which is summation of input” then I would clearly see that the second case should fail as multiplication (*) operator is not allowed. In designing above test cases, tester assumed that since addition is allowed and the multiplication is recurring addition, then he should get similar result by using multiplication (*) or addition (+) operator. Clearly, having a well document functional spec is absolutely necessary in writing effective negative test cases that could catch a bug.

2. **Negative test cases passing without invoking an error condition**: Test should be written so that they could invoke an error conditions. In our example Addition Application discussed above, test should be written so that 2*2 signals an error condition. This can be achieved by how you write the test. If in our example you are just comparing the result you won’t see it as an error but if you write test to check for invalid operator, you will see error.
3. **Negative test cases not handling specific error situation mentioned in specs:** This is another situation which is a slight variant of what was discussed above. If the negative test cases are written and they don’t hit the error condition a program is designed to handle, then that application is not getting tested for those error situation and we can not say for sure that application would be able to handle those special condition. Tests should be included in try..catch block and in catch block one should test for the exception returned by the program. If these exceptions are similar to what mentioned in the specs then the test is actually testing for that error condition. Designing such test cases may need developer help as they know how, where and when they would handle an exception in the code.

4. **Negative test cases with restrictive domain:** Don’t restrict the domain of your negative test cases to the functional specs. Imagine all sorts of things that can go wrong while writing negative test cases: memory corruption, stack overflow, security issue, resource conflict, data locking etc. Make sure that you have test that covers aspects not mentioned in the specs.

### 6. Regression Testing Tools

Test automation is a process of writing a computer program to do testing that would otherwise need to be done manually. Once tests have been automated, they can be run quickly and repeatedly. This is often the most cost effective method for software products that have a long maintenance life, because even minor patches over the lifetime of the application can potentially cause working functionality (at an earlier point in time) to break.

**Telerik TestStudio** is an all in one testing software for functional, load, performance and mobile app testing. The in-depth functional testing includes native web and desktop apps testing along with mobile and tablet apps, HTML5, AJAX, Silverlight and WPF apps testing. Additionally testing teams can rely on the product to test JavaScript calls, Telerik controls, dynamic page synchronization, client-side behaviors, UI virtualizations and XAML animations.

**QuickTest Professional** software provides functional and regression test automation for software applications and environments. HP QuickTest Professional supports keyword and scripting interfaces and features a graphical user interface. Its features are: a cascaded optimization system, the industry's deepest and broadest insight into IT-controlled assets, and a secure, comprehensive, operational environment for a hybrid world, enhanced expert view, business process testing, screen recorder etc.

**Watir**, pronounced water, is an open-source (BSD) family of ruby libraries for automating web browsers. It allows you to write tests that are easy to read and maintain. It is simple and flexible. It clicks links, fills in forms, and presses buttons. Watir also checks results, such as whether expected text appears on the page. Its features are: to connect to databases, read data files and spreadsheets, export XML, and structure your code as reusable libraries etc.

**TOSCA Testsuite** is a software tool for the automated execution of functional and regression software testing. In addition to test automation functions, it includes integrated test management, a graphical user interface (GUI), a command line interface (CLI) and an application programming interface (API), generation of dynamic, synthetic test data, highly automated business dynamic steering of test case generation and the unified handling and executing of manual and automated as well as GUI and non-GUI tests etc.

**Selenium** is a portable software testing framework for web applications. Selenium provides a record/playback tool for authoring tests without learning a test scripting language. It includes features like record and playback, intelligent field selection, Xpath as needed, auto complete for all common selenium commands, walk through tests, debug and set breakpoints, ruby scripts, or other formats, support for selenium user-extensions file, option to automatically assert the title of every page etc.
APPENDICES

Installation testing

Installation testing An installation test assures that the system is installed correctly and working at actual customer's hardware.

Compatibility testing

A common cause of software failure (real or perceived) is a lack of its compatibility with other application software, operating systems (or operating system versions, old or new), or target environments that differ greatly from the original (such as a terminal or GUI application intended to be run on the desktop now being required to become a web application, which must render in a web browser).

Smoke and sanity testing

Sanity testing determines whether it is reasonable to proceed with further testing. Smoke testing is used to determine whether there are serious problems with a piece of software, for example as a build verification test.

Regression testing

Regression testing focuses on finding defects after a major code change has occurred. Specifically, it seeks to uncover software regressions, or old bugs that have come back.

Acceptance testing

Acceptance testing can mean one of two things: A smoke test is used as an acceptance test prior to introducing a new build to the main testing process, i.e. before integration or regression. Acceptance testing performed by the customer, often in their lab environment on their own hardware, is known as user acceptance testing (UAT). Acceptance testing may be performed as part of the hand-off process between any two phases of development.

Alpha testing

Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.

Beta testing

The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users.

Functional

Functional testing refers to activities that verify a specific action or function of the code. These are usually found in the code requirements documentation, although some development methodologies work from use cases or user stories. Functional tests tend to answer the question of "can the user do this" or "does this particular feature work."

Non-functional testing

Non-functional testing refers to aspects of the software that may not be related to a specific function or user action, such as scalability or other performance, behavior under certain constraints, or security. Testing will determine the flake point, the point at which extremes of scalability or performance leads to unstable execution.

Destructive testing

Destructive testing attempts to cause the software or a sub-system to fail. It verifies that the software functions properly even when it receives invalid or unexpected inputs, thereby establishing the robustness of input validation and error-management routines.

Software performance testing
Performance testing is generally executed to determine how a system or sub-system performs in terms of responsiveness and stability under a particular workload. It can also serve to investigate, measure, validate or verify other quality attributes of the system, such as scalability, reliability and resource usage.

**Usability testing**

Usability testing is needed to check if the user interface is easy to use and understand. It is concerned mainly with the use of the application.

**Security testing**

Security testing is essential for software that processes confidential data to prevent system intrusion by hackers.
REFERENCES


