

THE REPUBLIC OF TURKEY

BAHCESEHIR UNIVERSITY

**REDUCING TEST AUTOMATION MAINTENANCE
COST BY CLASS-DRIVEN APPROACH**

Master's Thesis

SERKAN AKOĞLANOĞLU

İSTANBUL, 2012

**THE REPUBLIC OF TURKEY
BAHCESEHIR UNIVERSITY**

**GRADUATE SCHOOL OF NATURAL AND APPLIED
SCIENCES**

COMPUTER ENGINEERING

**REDUCING TEST AUTOMATION MAINTENANCE
COST BY CLASS-DRIVEN APPROACH**

Master's Thesis

SERKAN AKOĞLANOĞLU

Supervisor: Assist.Prof.Dr. M. Alper TUNGA

İSTANBUL, 2012

**THE REPUBLIC OF TURKEY
BAHCESEHIR UNIVERSITY**

**THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
COMPUTER ENGINEERING**

Name of the thesis :Reducing Test Automation Maintenance
Cost By Class-Driven Approach
Name/Last Name of the Student :Serkan Akođlanođlu
Date of Thesis Defense :11/06/2012
The thesis has been approved by the Graduate School of Natural and Applied Sciences.

Assoc. Prof. Tunç Bozbura
Graduate School Director
Signature

I certify that this thesis meets all the requirements as a thesis for the degree of Master of Arts.

Assist.Prof.Dr. Çađrı Gungör
Program Coordinator
Signature

This is to certify that we have read this thesis and we find it fully adequate in scope, quality and content, as a thesis for the degree of Master of Arts.

Examining Comittee Members

Signature

Thesis Supervisor
Assist.Prof.Dr. M. Alper TUNGA

Assist.Prof.Dr. Tefvik Aytekin

Assist.Prof.Dr. Yücel Batu Salman

ACKNOWLEDGEMENTS

I first wish to thank to my family and friends for their unlimited support and love in every stage of my life. I am sure that they patiently waited to see the steps in my academic career and I am very happy to make their dreams come true.

I also wish to thank Assist.Prof.Dr. M. Alper TUNGA for his encouragement and assistance in every related with the project. I cannot forget their support in this project and they have a great role in my success.

Last but not least, I wish to thank to all instructors of Bahçeşehir University and my friends for their patience, support and love.

ABSTRACT

Reducing Test Automation Maintenance Cost By Class-Driven Approach

Akođlanoglu, Serkan

M.S. Department of Computer Engineering

Supervisor: Assoc.Prof.Dr. M. Alper TUNGA

June 2012, 64 pages

The objective of this thesis is to reduce maintenance cost of test automation projects by using class-based architecture. Software test automation is a software development project. Thus, every test automation project should have maintenance phase. The format and behaviour of a system's user interface changes with each new release and so the automated tests must also change. Classical test automation tools save the movements of the user's on the application then provides these scenarios to be run automatically. Changes made on the system under test requires updates on the automated test scripts. In classical methods, these scenarios are re-recorded. On the class-based architecture, all system's user interface is covered by a class. Each change is handled from the class. And test scenarios are written by using the objects that are driven from related classes. Therefore re-recording of each scenario is not needed.

Keywords: Test Automation, Test Automation Maintenance Cost, Class-Driven Architecture.

ÖZET

Test Otomasyon Bakım Maliyetinin Sınıfa Dayalı Mimari ile Azaltılması

Akođlanođlu, Serkan

Yüksek Lisans, Bilgisayar Mühendisliđi Bölümü

Tez Yöneticisi: Yr.Doç.Dr. M. Alper TUNGA

Haziran 2012, 64 sayfa

Bu çalışmanın amacı test otomasyon projelerinde, sınıfa dayalı mimari kullanarak, bakım maliyetlerini klasik otomasyon yöntemlerine göre düşürmektir. Test otomasyonu da bir yazılım geliştirme projesidir. Bu nedenle her test otomasyon projesinin bakım sürecinde olmalıdır. Test edilen uygulamanın her yeni sürümünde kullanıcı arabirimde yapılan değişiklikler test otomasyon senaryolarında değişmesini gerektirir. Klasik test otomasyon araçları, kullanıcının uygulama üzerindeki hareketlerini bir senaryo olarak kaydedip, daha sonra bu senaryoların otomatik olarak çalıştırılmasını sağlar. Test edilen uygulamada yapılan değişiklikler, otomatik senaryoların değiştirilmesini gerektirir. Klasik yöntemde bu senaryo yeniden kaydedilir. Sınıfa dayalı mimari, test edilen uygulama üzerindeki kullanıcı arabirim nesnelere sınıf olarak tanımlar. Otomatik test senaryoları bu sınıflardan türetilen nesnelere yazılır. Test edilen uygulamada bir ekran üzerinde yapılan değişiklik, o sınıfa karşılık gelen test otomasyon sınıfı üzerinde yapılır. Test senaryosunu yeniden kaydetmek gerekmez.

Anahtar Kelimeler: Test Otomasyonu, Test Otomasyon Bakım Maliyeti, Sınıfa Dayalı Mimari

TABLE OF CONTENT

| | |
|--|-------------|
| FIGURES | viii |
| ABBREVIATIONS | ix |
| 1. INTRODUCTION | 1 |
| 1.1 PROBLEM DEFINITION | 2 |
| 1.2 SOLUTION | 2 |
| 2. LITERATURE SURVEY | 5 |
| 3. INTRODUCTION TO SOFTWARE TESTING | 8 |
| 3.1 WHAT IS SOFTWARE TESTING | 8 |
| 3.2 WHY SOFTWARE TESTING IS IMPORTANT | 8 |
| 3.3 SOFTWARE TESTING FUNDAMENTALS | 10 |
| 3.3.1 Testing Objectives | 10 |
| 3.3.2 Test Information Flow | 10 |
| 3.3.3 Test Case Design | 10 |
| 3.4 SOFTWARE TESTING METHODOLOGIES | 11 |
| 3.4.1 Waterfall Model | 11 |
| 3.4.2 V Model | 12 |
| 3.4.3 Spiral Model | 12 |
| 3.4.4 Rational Unified Process (RUP) | 12 |
| 3.4.5 Agile Model | 13 |
| 3.4.6 Rapid Application Development (RAD) | 13 |
| 3.5 SOFTWARE TESTING LIFE CYCLE | 13 |
| 3.6 SOFTWARE TESTING TECHNIQUES | 18 |
| 3.6.1 White Box Testing | 18 |
| 3.6.1.1 Advantages of White Box Testing | 18 |
| 3.6.1.2 Disadvantages of White Box Testing | 19 |
| 3.6.1.3 Types of Testing under White Box Testing Strategy | 19 |
| 3.6.2 Black Box Testing | 20 |
| 3.6.2.1 Types of Testing under Black Box Testing Strategy | 21 |

| | | |
|-----------|---|------------------------------|
| 3.7 | TYPES OF SOFTWARE TESTING..... | 22 |
| 3.8 | COST OF SOFTWARE TESTING AND QUALITY | 23 |
| 3.8.1 | ROI Of Software Testing..... | 24 |
| 4. | INTRODUCTION TO SOFTWARE TEST AUTOMATION | 25 |
| 4.1 | WHAT IS SOFTWARE TEST AUTOMATION | 25 |
| 4.1.1 | Testing And Test Automation Are Different | 25 |
| 4.2 | WHAT ARE THE ADVANTAGES OF TEST AUTOMATION | 26 |
| 4.3 | COMMON PROBLEMS OF TEST AUTOMATION..... | 27 |
| 4.3.1 | The Limitations of Automating Software Testing..... | 29 |
| 5. | A FRAMEWORK MODEL FOR THE SUCCESS OF THE TEST AUTOMATION | 31 |
| 5.1 | PROBLEMS WITH TEST AUTOMATION | 31 |
| 5.2 | FROM RECORD/PLAYBACK TO FRAMEWORKS..... | 31 |
| 5.2.1 | What Is The Problem With Record/Playback Approach | 31 |
| 5.2.2 | Types Of Test Automation Frameworks | 32 |
| 5.2.2.1 | Data Driven Test Frameworks | 32 |
| 5.2.2.1.1 | Advantage of Data Driven Automation Test Framework .. | 33 |
| 5.2.2.2 | Keyword Driven Test Frameworks | 33 |
| 5.2.2.2.1 | Advantages of Keyword Driven Framework..... | 33 |
| 5.2.2.3 | Hybrid Test Frameworks | 34 |
| 5.3 | CLASS-DRIVEN FRAMEWORK APPROACH..... | 34 |
| 6. | CLASS-DRIVEN MODEL FOR MAINTAINABLE TEST AUTOMATION | 35 |
| 6.1 | CONCEPT OF CLASS IN GENERAL..... | 35 |
| 6.2 | ADVANTAGES OF OBJECT ORIENTED BASIC CONCEPT | 36 |
| 6.3 | WHAT DOES TEST CLASS MEAN IN CLASS-DRIVEN MODEL | 36 |
| 6.4 | THE BENEFIT OF THE MODEL | 37 |
| 6.5 | STRUCTURE OF CLASS-DRIVEN APPROACH | 40 |
| 6.6 | HOW TO HANDLE CHANGES BY CLASS-DRIVEN APPROACH..... | 49 |
| 7. | A CASE STUDY | 52 |
| 8. | CONCLUSION AND FUTURE WORKS | 60 |
| | REFERENCES | Error! Bookmark not defined. |

FIGURES

| | |
|---|----|
| Figure 1.1: Software Testing Lifecycle..... | 3 |
| Figure 6.1: An Application Form | 37 |
| Figure 6.2: Main Page of Application..... | 41 |
| Figure 6.3: Finance Modul in ERP..... | 43 |
| Figure 6.4: Payment Page | 44 |
| Figure 6.5: ERP Form | 47 |
| Figure 6.6: Finance Modul..... | 47 |
| Figure 6.7: Payment Section..... | 48 |
| Figure 6.8: Change on the Payment | 50 |
| Figure 7.1: QTP Recording and Converting Screen | 53 |
| Figure 7.2: QTP Object Repository..... | 54 |
| Figure 7.3: Test Automation Maintenance Effort With Classic Method..... | 55 |
| Figure 7.4: Test Automation Maintenance Effort With Class-Driven Architecture | 56 |
| Figure 7.5: A Test Script written with Class-Driven | 57 |
| Figure 7.6: A Test Class..... | 58 |

ABBREVIATIONS

| | |
|------|---|
| SQA | : Software Quality Assurance |
| SDLC | : Software Development Life Cycle |
| SRS | : Software Requirement Specification |
| GUI | : Graphical User Interface |
| AUT | : Application Under Test |
| UI | : User Interface |
| OOP | : Object-Oriented Programming |
| GSM | : Global System for Mobile Communications |
| VB | : Visual Basic |

1. INTRODUCTION

I am trying to write a thesis about Software Test Automation Maintenance. I will analyze the current situation of software test description. I've had the opportunity to work in a group who writes automated tests using applications to ensure compatibility in future releases of applications, and to find bugs.

Software tests are as valuable as source code to a software project. Over the long term, maintainable software tests significantly lower a project's cost. However it is very difficult to write maintainable software tests, especially executable ones.

Much of the cost of software development is maintenance, changing the software after it is written. Test teams in many organizations that attempted test automation abandon the effort within a few months. The most common reason is that the tests quickly become brittle and too costly to maintain. The slightest change in the implementation of the system requires to change much of the test scripts. For example, renaming a button, breaks swarms of tests, and fixing the tests is too time consuming.

The requirements are constantly changing due to the changes on business rules therefore production code changes. In order to increase efficiency of the test phase, test cases should be up to date even manual or automated as the requirements change. It is expected that test automation efforts would be more likely to succeed than in years past. Test teams have no trouble creating thousands of automated regression tests since vendors market very capable tools. However, teams spend most of their time maintaining the automated test scripts. The little change in the system requires fixing the test scripts. Generally project leads decide to give up the test automation since maintenance time of automated test cases is more than running the cases manually.

1.1 PROBLEM DEFINITION

The need to change tests comes from two directions: changes in requirements and changes in the system's implementation. Either kind of change can break any number of automated tests. If the tests become out of sync with either the requirements or the implementation, people stop running the tests or stop trusting the results. To get the tests back in sync, we must change the tests to adapt to the new requirements or the new implementation. But the problem is, maintenance cost for ensuring the synchronization of test scripts, more than expected, and this usually ends with the failure of the test automation. Test teams have to abandon test automation due to the unexpected maintenance costs.

The following problems are often encountered during test automation projects and may result in failing test automation projects:

- i. Management doesn't treat test automation as a software development. Anyone can test and automation is easy, just record and playback.
- ii. QA team select wrong set of test cases for test automation. In the same time, management aims for 100 percent of test automation of all test cases.
- iii. QA Engineers spend time between manual and automation testing, instead of concentrating on one task.
- iv. No one realized that automation test cases are difficult to maintain and manage.
- v. The development, maintenance and management of the automated test scripts often need additional time and resources than manual test execution by inexperienced testers.

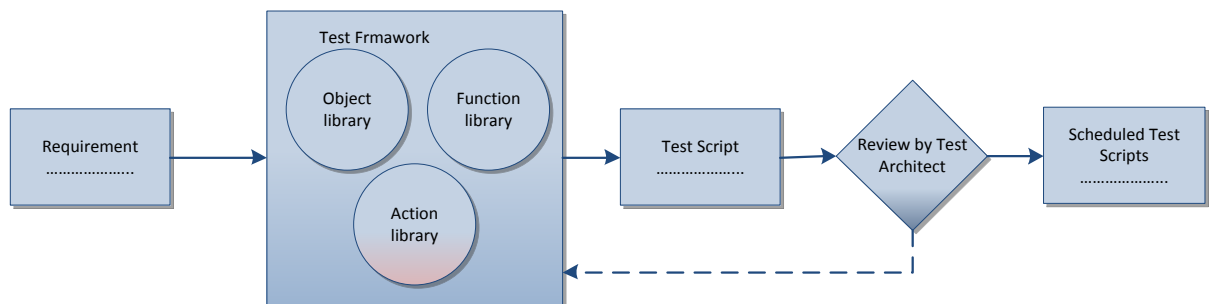
1.2 SOLUTION

I want to express our pilot test automation project which needs more effort than estimated at the begin. I wrote about 400 test scripts and covered nearly all the requirements. Each week, we release new version and I run all the automated cases before production. But I see that I spend 40 percent of my test effort to maintain available automated test scripts since production code changes and those scripts have to be updated accordingly. Due to the increasing cost of test automation, I decided to review our automation process. I aimed to write test scripts that of maintenance cost not to exceed 10 percent of all test cost.

My experiences proved that the most common reason for the failure of the test automation maintenance projects is not considering these as a software project. Automation is more than “Record and Playback” actions.

All the rules we know about writing software also applies to test automation. Test scripts should be designed by applying at least simple coding principles such as “Don’t Repeat Yourself”. Since source codes change every release, test scripts should be updated accordingly. Anytime we need to update a test automation script, we only have to change one module, class, include or macro, perhaps only a variable value.

Figure 1.1: Software Testing Lifecycle



We use test framework so that we are not surprised with the maintenance cost of test automation. Automation framework is nothing but the procedure and overall structure we followed for using automation tools. We are following Hybrid automation framework. Hybrid framework is the combination of both keyboard driven and data driven frameworks. About implementation of framework we need to take care of all test objects those are not going to be changed in application should be hardcoded and most frequently changing objects should be taken from the data sheet. For example, we create a object library and a function library, and put reusable test objects and functions to there. When we maintain a test script we try to make the changes on the common libraries as much as possible. We try to structure common test libraries as they pair with the production code and also requirements. When we write a test automation script, we send it to our test architects to review it. They decide if it is maintainable or not.

To summarize the above, according to the test automation framework we build;

- i. Test Automation is software engineering.

- ii. Record/Playback automation reduces limits of maintainability.

At the beginning of the test automation projects, maintenance costs should be considered as like software projects.

2. LITERATURE SURVEY

At the moment I read pretty much about the different automation frameworks, but I may not have the time to review all of them. So I plan to read about them and make the thesis more literature - based

- James Bach's "Software Test Automation Snake Oil", 1997
- Kaner, Bach and Pettichord's book "Lessons Learned in Software Testing", 2002
- Achieving the Full Potential of Software Test Automation, Whitepaper by LogiGear, 2004
- Improving the Maintainability of Automated Test Suites, Cem Kaner, 1997
- Writing Maintainable Automated Acceptance Tests, Dale H. Emery, 2009
- APM tools: Applying automated testing earlier in the development lifecycle, Kevin Parker
- Automated Software Testing - A Perspective ¹
- Getting Automated Testing Under Control, 1999, STQE Magazine
- Useful Automated Software Testing Metrics By Thom Garrett IDT, LLC
- The minefield analogy ²
- The papers of Doug Hoffman on test automation ³
- The anti-intellectualism pushed by some proponents of the blackbox test automation community
- Kaner's Black Box Software Testing Course, Cem Kaner, 2003
- Context Driven Software Testing, Cem Kaner, 2002
- Jon Kohl's work on "Man and Machine", or the cyborg approach (instead of computer-alone test execution and evaluation)

¹ <http://www.testingstuff.com/autotest.html>

² <http://www.testingperspective.com/tpwiki/doku.php?id=minefield>

³ <http://www.softwarequalitymethods.com/H-Papers.html>

A paper presented at Quality Week'97 by Cem Kaner, describes test automation as a software development. Record-play tools can help writing automated test scripts but they don't give a methodology. Kaner's paper explain the framework-driven approach.

Whitepaper by LogiGear, 2004 : Provide definitions for several testing terms and concepts. This paper will first discuss the key benefits of software test automation, and examine the most common implementation tactics. It will then analyze the key reasons why test automation efforts often fail to meet their potential. Finally, it will show how using a keyword-based test automation strategy enables organizations to avoid those problems.

According to Cem Kaner, Test automation can add a lot of complexity and cost to a test team's effort, but it can also provide some valuable assistance if its done by the right people, with the right environment and done where it makes sense to do so. I hope by sharing some pointers that I feel are important that you'll find some value that translates into saved time, money and less frustration in your efforts to implement test automation back on the job.

Machiel van der Bijl of the University of Twente has developed a software package that eliminates the need for manual software testing. If his system takes off, this could represent an enormous cost and time savings for software developers. In particular, I'm impressed that the software doesn't just run the tests, it actually develops them. Mr. Van der Bijl has also started a spin-off company, Axini, to market the process.

According to Thom Garrett, as part of a successful automated testing program it is important that goals and strategies are defined and then implemented. During implementation progress against these goals and strategies set out to be accomplished at the onset of the program needs to be continuously tracked and measured. This article discusses various types of automated and general testing metrics that can be used to measure and track progress. Based on the outcome of these various metrics the defects remaining to be fixed in a testing cycle can be assessed; schedules can be adjusted accordingly or goals can be reduced. For example, if a feature is still left with too many high priority defects a decision can be made that the ship date is moved or that the system is shipped or even goes live without that specific feature.

A 2011 paper by Artzi et al, describes rudimentary test automation framework for perform feedback-directed test generation for JavaScript web applications. That paper presented the case study of a specific system (rather than general or all web applications) also it needed access to AUT's source code.

Another 2011 paper by James M. Slack described a test automation framework using "AutoIt" tool and an Excel Sheet as a Data Container for test data. The framework proposed was tool specific and the disadvantages of using excel sheet as a data

container were not covered, as excel sheet representation is not suitable for more complex and dynamic web applications.

A 2010 paper by Manpreet Kaur et al “Xml Schema Based Approach for Testing of Software Components” mentions apt use of XML format for representing test data. Another 2010 paper “DEVELOPMENT OF TEST AUTOMATION FRAMEWORK FOR TESTING AVIONICS SYSTEMS” describes aptly some basics for implementing data driven frameworks but again it does not give a generic model or architecture for general design.

A 2002 paper by Tsai, Paul, Song, and Cao presented a description of an XML based framework named Coyote which was designed for testing web services. Again, this paper was a case study and presented no conclusions.

A 2009 paper by Merchant, Tellez, and Venkatesan presented a browser agnostic UI test framework for web applications and concluded that using the framework reduced the time required toInternational Journal of Computer Applications create test case scenarios by 50 percent compared to a manual approach .

Kaner supposed that the myth that says people with little programming experience can use test tools to quickly create extensive test suites, is spread by tool vendors. The tools are easy to use. Maintenance of the test suites not a problem. Therefore, the story goes, a development manager can save lots of money and aggravation, and can ship software sooner, by using one of these tools to replace some of the pesky testers. But several companies have failed to use these tools effectively.

Tests, especially automated software tests, represent significant investment. Windows NT 5.0 reportedly will have 48 million lines of source code, while associated test code consists of 7.5 million lines of source code. In this case, roughly one seventh of programming resources are spent in test automation work. Window NT 5.0 is going to be maintained over the next several years; maintainable tests would make a big difference in long-term costs.

3. INTRODUCTION TO SOFTWARE TESTING

3.1 WHAT IS SOFTWARE TESTING

Testing is a process of evaluating a particular product to determine whether the product contains any defect.

Software Testing is a process of evaluating a system by manual or automatic means and verify that it satisfies specified requirements or identify differences between expected and actual results.

Software Testing is the process used to help identify the correctness, completeness, security, and quality of developed computer software. Testing is a process of technical investigation, performed on behalf of stakeholders, that is intended to reveal quality-related information about the product with respect to the context in which it is intended to operate. This includes, but is not limited to, the process of executing a program or application with the intent of finding errors. Quality is not an absolute; it is value to some person. With that in mind, testing can never completely establish the correctness of arbitrary computer software; testing furnishes a criticism or comparison that compares the state and behaviour of the product against a specification. An important point is that software testing should be distinguished from the separate discipline of Software Quality Assurance (SQA), which encompasses all business process areas, not just testing.

Software must be tested to have confidence that it will work as it should in its intended environment.

3.2 WHY SOFTWARE TESTING IS IMPORTANT

Software testing is performed to verify that the completed software package functions according to the expectations defined by the requirements/specifications. The overall objective is not to find every software bug that exists, but to uncover situations that could negatively impact the customer, usability and/or maintainability.

Test provides the followings;

- i. To discover defects.
- ii. To avoid user detecting problems.
- iii. To prove that the software has no faults.
- iv. To learn about the reliability of the software.
- v. To avoid being sued by customers.

- vi. To ensure that product works as user expected.
- vii. To stay in business.
- viii. To detect defects early, which helps in reducing the cost of defect fixing.

In March 1992, a man living in Newtown near Boston, Massachusetts, received a bill for his as yet unused credit card stating that he owed \$0.00. He ignored it and threw it away.

In April, he received another and threw that one away too.

The following month, the credit card company sent him a very nasty note stating they were going to cancel his card if he didn't send them \$0.00 by return of post. He called them and talked to them; they said it was a computer error and told him they'd take care of it.

The following month, our hero decided that it was about time that he tried out the troublesome credit card figuring that if there were purchases on his account it would put an end to his ridiculous predicament. However, in the first store that he produced his credit card in payment for his purchases, he found that his card had been cancelled.

He called the credit card company who apologized for the computer error once again and said that they would take care of it. The next day he got a bill for \$0.00 stating that payment was now overdue. Assuming that, having spoken to the credit card company only the previous day, the latest bill was yet another mistake, he ignored it, trusting that the company would be as good as their word and sort the problem out.

The next month, he got a bill for \$0.00 stating that he had 10 days to pay his account or the company would have to take steps to recover the debt.

Finally giving in, he thought he would play the company at their own game and mailed them a cheque for \$0.00. The computer duly processed his account and returned a statement to the effect that he now owed the credit card company nothing at all.

A week later, the man's bank called him asking him what he was doing writing a cheque for \$0.00. After a lengthy explanation, the bank replied that the \$0.00 cheque had caused their cheque processing software to fail.

The bank could now not process ANY cheques from ANY of their customers that day because the cheque for \$0.00 was causing the bank's computer to crash.

The following month, the man received a letter from the credit card company claiming that his cheque had bounced and that he now owed them \$0.00 and unless he sent a cheque by return of post they would be taking steps to recover the debt.

The man, who had been considering buying his wife a computer for her birthday, bought her a typewriter instead.

3.3 SOFTWARE TESTING FUNDAMENTALS

During testing, the software engineering produces a series of test cases that are used to cut the software into pieces. Testing is the one step in the software process that can be seen by the developers as destructive instead of constructive. Software engineers are typically constructive people and testing requires them to overcome prejudiced concepts of correctness and deal with conflicts when errors are identified.

3.3.1 Testing Objectives

A number of rules that act as testing objectives are:

- i. Testing is process of executing a problem with the purpose of finding errors.
- ii. A good test case will have a good chance of findings an undiscovered error.
- iii. A successful test case uncovers a new error.

3.3.2 Test Information Flow

Tests are performed and all outcomes considered test results are compared with expected results. When erroneous data is identified, error is implied and debugging begins. The debugging procedure is the most unpredictable element of the testing procedure. An error that indicates a discrepancy of 0.01 percent between the expected and the actula results can take hours, days or months to identify and correct. It is the uncertainty in debugging that causes testing to be difficult to schedule reliability.

3.3.3 Test Case Design

The design of software testing can be a challenging process. However software engineers often see testing as an afterthought, producing test cases that feel right but have little assurance that they are complete. The objective of testing is to have the highest likelihood of finding the most errors with a minimum amount of timing and effort. A large number of test case design methods have been developed that offer the developer with a systematic approach to testing. Methods offer an approach that can ensure the completeness of tests and offer the highest likelihood for uncovering errors in software.

An engineering product can be tested in two ways:

- i. Knowing the specified functions that the product has been designed to perform, tests can be performed that show that each function is fully operational.
- ii. Knowing the internal workings of a product, tests can be performed to see if they jell.

The first test approach is known as a black box testing and the second one white box testing.

3.4 SOFTWARE TESTING METHODOLOGIES

These are some of the commonly used test methodologies:

- i. Waterfall model
- ii. V model
- iii. Spiral model
- iv. RUP
- v. Agile model
- vi. RAD

Let us have a look at each of these methodologies one by one.

3.4.1 Waterfall Model

The waterfall model adopts a 'top down' approach regardless of whether it is being used for software development or testing. The basic steps involved in this software testing methodology are as follows:

- i. Requirement analysis
- ii. Test case design
- iii. Test case implementation
- iv. Testing, debugging and validating the code or product
- v. Deployment and maintenance

In this methodology, you move on to the next step only after you have completed the present step. There is no scope for jumping backward or forward or performing two steps simultaneously. Also, the model follows a non-iterative approach. The main benefit of this methodology is its simplistic, systematic and orthodox approach. However, it has many shortcomings since bugs and errors in the code are not discovered until and unless the testing stage is reached. This can often lead to wastage of time, money and other valuable resources.

3.4.2 V Model

The V model gets its name from the fact that graphical representation of different test process activities involved in this methodology resemble the letter 'V'. Basic steps involved in this methodology are more or less the same as in waterfall model. However, this model follows both a 'top-down' as well as 'bottom-up' approach (you can visualize them forming the letter 'V'). The benefit of using this methodology is that, both the development and testing activities go hand-in-hand. For example, as the development team goes about its requirement analysis activities, the testing team simultaneously begins with its acceptance testing activities. By following this approach, time delays are minimized and optimum utilization of resources assured.

3.4.3 Spiral Model

As the name implies, spiral model follows an approach in which there are a number of cycles (or spirals) of all the sequential steps of the waterfall model. Once the initial cycle gets completed, a thorough analysis and review of the achieved product or output is performed. If it is not as per the specified requirements or expected standards, a second cycle follows, and so on. This methodology follows an iterative approach and is generally suited for large projects having complex and constantly changing requirements.

3.4.4 Rational Unified Process (RUP)

The RUP methodology is also similar to the spiral model in the sense that entire testing procedure is broken up into multiple cycles or processes. Each cycle consists of four phases namely; inception, elaboration, construction and transition. At the end of each cycle, the product/output is reviewed and a further cycle (made up of the same four

phases) follows if necessary. Today, you will find certain organizations and companies adopting a slightly modified version of the RUP, which goes by the name, Enterprise Unified Process (EUP).

3.4.5 Agile Model

This methodology follows neither a purely sequential approach nor a purely iterative approach. It is a selective mix of both approaches in addition to quite a few and new developmental methods. Fast and incremental development is one of the key principles of this methodology. The focus is on obtaining quick, practical and visible outputs rather than merely following the theoretical processes. Continuous customer interaction and participation is an integral part of the entire development process.

3.4.6 Rapid Application Development (RAD)

The name says it all. In this case, the methodology adopts a rapid developmental approach by using the principle of component-based construction. After understanding the different requirements of the given project, a rapid prototype is prepared and is then compared with the expected set of output conditions and standards. The necessary changes and modifications are made following joint discussions with the customer or development team (in the context of software testing). Though this approach does have its share of advantages, it can be unsuitable if the project is large, complex and happens to be extremely dynamic in nature, wherein requirements change constantly.

3.5 SOFTWARE TESTING LIFE CYCLE

The software testing life cycle consists of various testing activities that need to be carried out to validate if the software meets the required design specification.

In every organization testing is an important phase in the development of a software product. However, the way it is carried out differs from one organization to another. It is advisable to carry out the testing process from the initial stages, with regard to the Software Development Life Cycle or SDLC to avoid any complications (Harshada Kekare, 2011).

Software testing has its own life cycle that meets every stage of the SDLC. The software testing life cycle diagram can help one understand its various phases. They are:

- i. Requirement Stage
- ii. Test Planning
- iii. Test Analysis
- iv. Test Design
- v. Test Verification and Construction
- vi. Test Execution
- vii. Result Analysis
- viii. Bug Tracking
- ix. Reporting and Rework
- x. Final Testing and Implementation
- xi. Post Implementation

Requirement Stage:

This is the initial stage of the software testing life cycle process. In this phase the developers take part in analyzing the requirements for designing a product. The role of software testers is also necessary in this phase as they can think from the 'users' point of view which the developers may not. Thus a team of developers, testers and users can be formed, to analyze the requirements of the product. Formal meetings of the team can be held in order to document the requirements which can further be used as software requirements specification or SRS.

Test Planning:

Test planning means to predetermine a plan well in advance to reduce further risks. A well-designed test plan document plays an important role in achieving a process-oriented approach. Once the requirements of the project are confirmed, a test plan is documented. The test plan structure is as follows:

- i. **Introduction:** This describes the objective of the test plan.
- ii. **Test Items:** The items that are required to prepare this document will be listed here such as SRS, project plan.

- iii. **Features to be tested:** This describes the coverage area of the test plan, that is, the list of features to be tested; that are based on the implicit and explicit requirements from the customer.
- iv. **Features not to be tested:** The incorporated or comprised features that can be skipped from the testing phase are listed here. Features that are out of scope of testing, like incomplete modules or those on low severity, for example, GUI features that don't hamper the process can be included in the list.
- v. **Approach:** This is the test strategy that should be appropriate to the level of the plan. It should be in acceptance with the higher and lower levels of the plan.
- vi. **Item pass/fail criteria:** Related to the show stopper issue. The criteria used has to explain which test item has passed or failed.
- vii. **Suspension criteria and resumption requirements:** The suspension criteria specifies the criteria that is to be used to suspend all or a portion of the testing activities, whereas resumption criteria specifies when testing can resume with the suspended portion.
- viii. **Test deliverable:** This includes a list of documents, reports, charts that are required to be presented to the stakeholders on a regular basis during the testing process and after its completion.
- ix. **Testing tasks:** This phase lists the testing tasks that need to be performed. This includes conducting the tests, evaluating the results and documenting them based on the test plan designed. This also helps users and testers to avoid incomplete functions and prevent waste of resources.
- x. **Environmental needs:** The special requirements of the test plan depending on the environment in which the application has to be designed are listed here.
- xi. **Responsibilities:** This phase assigns responsibilities to people who can be held responsible in case of a risk.
- xii. **Staffing and training needs:** Training on the application/system and on the testing tools to be used needs to be explained to the staff members who are responsible for the application.
- xiii. **Risks and contingencies:** This emphasizes on the probable risks and various events that can occur and what can be done in such situations.
- xiv. **Approval:** This decides who can approve the process as complete and allow the project to proceed to the next level that depends on the level of the plan.

Test Analysis:

Once the test plan documentation is done, the next stage is to analyze what types of software testing should be carried out at the various stages of SDLC.

Test Design:

Test design is done based on the requirements of the project documented in the SRS. This phase decides whether manual or automated testing is to be done. In automation testing, different paths for testing are to be identified first and writing of scripts has to be done if required. An end-to-end checklist that covers all the features of the project is necessary in the test design process.

Test Verification and Construction:

In this phase, the test plan, test design and automated test script are completed. Stress and performance testing plans are also completed at this stage. When the development team is done with a unit of code, the testing team is required to help them in testing that unit and report any bug in the product, if found. Integration testing and bug reporting is done in this phase of software testing life cycle.

Test Execution:

Planning and execution of various test cases is done in this phase. Once the unit testing is completed, the functionality of the tests is done in this phase. At first, top-level testing is done to find out the top-level failures and bugs are reported immediately to the development team to get the required workaround. Test reports have to be documented properly and the bugs have to be reported to the development team.

Result Analysis:

After the successful execution of the test case, the testing team has to retest it to compare the expected values with the actual values, and declare the result as pass/fail.

Bug Tracking:

This is one of the important stages as the Defect Profile Document (DPD) has to be updated for letting the developers know about the defect. Defect Profile Document contains the following:

- i. **Defect Id:** Unique identification of the Defect.
- ii. **Test Case Id:** Test case identification for that defect.
- iii. **Description:** Detailed description of the bug.

- iv. **Summary:** This field contains some keyword information about the bug, which can help in minimizing the number of records to be searched.
- v. **Defect Submitted By:** Name of the tester who detected/reported the bug.
- vi. **Date of Submission:** Date at which the bug was detected and reported.
- vii. **Build No:** Number of test runs required.
- viii. **Version No:** The version information of the software application in which the bug was detected and fixed.
- ix. **Assigned To:** Name of the developer who is supposed to fix the bug.
- x. **Severity:** Degree of severity of the defect.
- xi. **Priority:** Priority of fixing the bug.
- xii. **Status:** This field displays current status of the bug.

Reporting and Rework:

Testing is an iterative process. The bug that is reported and fixed by the development team, has to undergo the testing process again to assure that the bug found has been resolved. Regression testing has to be done. Once the Quality Analyst assures that the product is ready, the software is released for production. Before release, the software has to undergo one more round of top-level testing. Thus testing is an ongoing process.

Final Testing and Implementation:

This phase focuses on the remaining levels of testing, such as acceptance, load, stress, performance and recovery testing. The application needs to be verified under specified conditions with respect to the SRS. Various documents are updated and different matrices for testing are completed at this stage of the software testing life cycle.

Post Implementation:

Once the test results are evaluated, the recording of errors that occurred during the various levels of the software testing life cycle, is done. Creating plans for improvement and enhancement is an ongoing process. This helps to prevent similar problems from occurring in the future projects. In short, planning for improvement of the testing process for future applications is done in this phase.

3.6 SOFTWARE TESTING TECHNIQUES

Testing techniques can be used to effectively design efficient test cases. These techniques can be grouped into black-box and white-box techniques.

3.6.1 White Box Testing

White box testing strategy deals with the internal logic and structure of the code. The tests that are written based on the white box testing strategy incorporate coverage of the code written, branches, paths, statements and internal logic of the code. In order to implement white box testing, the tester has to deal with the code. White box test also needs the tester to look into the code and find out which unit/statement/chunk of the code is malfunctioning.

In other words, it is imperative that the tester has 'structural' knowledge about how the system has been implemented. Not only the code, but even the data flow and control flow have to be assessed. The **areas of the code**, that are tested using white box testing are:

- i. Code Coverage
- ii. Segment Coverage
- iii. Branch Coverage
- iv. Condition Coverage
- v. Loop Coverage
- vi. Path Testing
- vii. Data Flow Coverage

3.6.1.1 Advantages of White Box Testing

- i. As the knowledge of internal coding structure is prerequisite, it becomes very easy to find out which type of input/data can help in testing the application effectively.

- ii. Yet another advantage of white box testing is that it helps in optimizing the code.
- iii. It helps in removing the extra lines of code, which can introduce defects in the code.

3.6.1.2 Disadvantages of White Box Testing

- i. As knowledge of code and internal structure is a prerequisite, a skilled tester is needed to carry out this type of testing, and this, in turn, increases the cost of the software.
- ii. It is nearly impossible to look into every bit of code to find out hidden errors, which may create problems, resulting in failure of the application.

3.6.1.3 Types of Testing under White Box Testing Strategy

Unit Testing:

The developer carries out unit testing in order to check if the particular module or unit of code is working fine. The unit testing comes at the very basic level as it is carried out as and when the unit of the code is developed or a particular functionality is built.

Static and Dynamic Analysis:

While static analysis involves going through the code in order to find out any possible defect in the code, dynamic analysis involves executing the code and analyzing the output.

Statement Coverage:

In this type of testing, the code is executed in such a manner that every statement of the application is executed at least once. It helps in assuring that all the statements are executed without any side effect. Different coverage management tools are used to assess the percentage of the executable elements, which are currently been tested. (These tools are used for both statement as well as branch coverage.)

Branch Coverage:

No software application can be written in a continuous mode of coding. At some point we need to branch out the code in order to perform a particular functionality. Branch coverage testing helps in validating of all the branches in the code, and helps make sure that no branching leads to abnormal behavior of the application.

Memory Leak Testing:

When a code is written, there is a possibility that there is a problem of memory leak in the code, which makes the code faulty. Therefore, during the white box testing phase the code is tested to check, if there is memory leak in the code. In case of memory leak, more memory is required for the software and this affects the speed of the software making it slow.

Security Testing:

Security testing is carried out in order to find out how well the system can protect itself from unauthorized access, hacking (cracking, any code damage, etc.) which deals with the code of application. This type of testing needs sophisticated testing techniques.

Mutation Testing:

It is a kind of testing in which, the application is tested for the code that was modified after fixing a particular bug/defect. It also helps in finding out which code and which strategy of coding can help in developing the functionality effectively.

3.6.2 Black Box Testing

As the name "black box" suggests, no knowledge of internal logic or code structure is required. The types of testing under this strategy are totally based/focused on the testing for requirements and functionality of the work product/software application.

The base of the black box testing strategy lies in the selection of appropriate data as per functionality and testing it against the functional specifications in order to check for normal and abnormal behavior of the system. Nowadays, it is becoming common to route the testing work to a third party as the developer of the system knows too much of the internal logic and coding of the system, which makes it unfit to test the application by the developer.

3.6.2.1 Types of Testing under Black Box Testing Strategy

Functional Testing:

In this type of testing, the software is tested for the functional requirements. The tests are written in order to check if the application behaves as expected.

Stress Testing:

The application is tested against heavy load such as complex numerical values, large number of inputs, large number of queries, etc. which checks for the stress/load that the applications can withstand.

Load Testing:

The application is tested against heavy loads or inputs such as testing of websites in order to find out at what point the website/application fails or at what point its performance degrades.

Ad-hoc Testing:

This type of testing is done without any formal test plan or test case creation. Ad-hoc testing helps in deciding the scope and duration of the other testing methods and it also helps testers in learning the application prior to starting with any other testing.

Exploratory Testing:

This testing is similar to the ad-hoc testing and is done in order to learn/explore the application.

Usability Testing:

This testing is also called 'Testing for User-Friendliness'. This testing is done if user interface of the application stands an important consideration and needs to be specific for the specific type of user.

Smoke Testing:

This type of testing is also called sanity testing and is done in order to check if the application is ready for further major testing and is working properly without failing up to least expected level.

Recovery Testing:

Recovery testing is basically done in order to check how fast and better the application can recover against any type of crash or hardware failure, etc. Type or extent of recovery is specified in the requirement specifications.

Volume Testing:

Volume testing is done against the efficiency of the application. Huge amount of data is processed through the application (which is being tested) in order to check the extreme limitations of the system.

User Acceptance Testing:

In this type of testing, the software is handed over to the user in order to find out if the software meets the user expectations and works as it is expected to.

Alpha Testing:

In this type of testing, the users are invited at the development center where they use the application and the developers note every particular input or action carried out by the user. Any type of abnormal behavior of the system is noted and rectified by the developers.

Beta Testing:

In this type of testing, the software is distributed as a beta version to the users and users test the application at their sites. As the users explore the software, in case if any exception/defect occurs, then that is reported to the developers.

3.7 TYPES OF SOFTWARE TESTING

Various software testing methodologies guide you through the consecutive software testing types. To determine the true functionality of the application being tested, test cases are designed to help the developers. Test cases provide you with the guidelines for going through the process of testing the software. Software testing includes two basic types of software testing, Manual Scripted Testing and Automated Testing.

- Manual Scripted Testing: This is considered to be one of the oldest type of software testing methods, in which test cases are designed and reviewed by the team, before executing it.

- Automated Testing: This software testing type applies automation in the testing, which can be applied to various parts of a software process such as test case management, executing test cases, defect management, reporting of the bugs/defects. The bug life cycle helps the tester in deciding how to log a bug and also guides the developer to decide on the priority of the bug depending upon the severity of logging it. Software bug testing or software testing to log a bug, explains the contents of a bug that is to be fixed. This can be done with the help of various bug tracking tools such as Bugzilla and defect tracking management tools like the Test Director.

3.8 COST OF SOFTWARE TESTING AND QUALITY

The title of Phil Crosby's book says it all: *Quality Is Free*. Why is quality free? Like Crosby and J.M. Juran, Jim Campenella, in *Principles of Quality Costs*, illustrates a technique of analyzing the costs of quality start by breaking down these costs as follows:

$$C_{\text{quality}} = C_{\text{conformance}} + C_{\text{nonconformance}}$$

Conformance costs include prevention costs and appraisal costs. Prevention costs include money spent on quality assurance—tasks like training, requirements and code reviews, and other activities that promote good software.

Appraisal costs include money spent on planning test activities, developing test cases and data, and executing those test cases once.

Nonconformance costs come in two flavors, internal failures and external failures. The costs of internal failure include all expenses that arise when test cases fail the first time they're run, as they often do. A programmer experiences a cost of internal failure while debugging problems found during her own unit and component testing.

Once we get into formal testing in an independent test team, the costs of internal failure increase. Think through the process: The tester researches and reports the failure, the programmer finds and fixes the fault, the release engineer produces a new release, the system administration team installs that release in the test environment, and the tester retests the new release to confirm the fix and to check for regression. (Rex Black, 2000)

The costs of external failure are those incurred when, rather than a tester finding a bug, the customer does. These costs will be even higher than those associated with either kind of internal failure, programmer-found or tester-found. In these cases, not only does the same process described for tester-found bugs occur, but you also incur the technical support overhead and the more expensive process of releasing a fix to the field rather

than to the test lab. In addition, consider the intangible costs: Angry customers, damage to the company image, lost business, and maybe even lawsuits.

Two observations lay the foundation for the enlightened view of testing as an investment. First, like any cost equation in business, we will want to minimize the cost of quality. Second, while it is often cheaper to prevent problems than to repair them, if we must repair problems, internal failures cost less than external failures.

3.8.1 ROI Of Software Testing

Many project managers are aware of the value of testing in their projects, and understand that investing in the testing phase is an investment in the quality of their project as a whole. However, there are still some that view testing as that little bit that happens at the end after everything else is done: when the testers do a few quick checks just to make sure everything works.

4. INTRODUCTION TO SOFTWARE TEST AUTOMATION

4.1 WHAT IS SOFTWARE TEST AUTOMATION

Automated software testing is the process of creating test scripts that can then be run automatically, repetitively, and through many iterations. Done properly, automated software testing can help to minimize the variability of results, speed up the testing process, increase test coverage (the number of different things tested), and ultimately provide greater confidence in the quality of the software being tested.

Automating software testing can significantly reduce the effort required for adequate testing, or significantly increase the testing which can be done in the limited time. Test can be run in minutes that would take hours to run manually.

4.1.1 Testing And Test Automation Are Different

Testing is skill. For any system there is an astronomical number of possible test cases and yet practically we have time to run only a very small number of them. Yet this small number of the test cases is expected to find most of the defects in the software, so the job of selecting which test cases to build and run is an important one. Selecting test cases at random is not an effective approach to testing. A more thoughtful approach is required if good test cases are to be developed. (Mark Fewster, 1999)

What exactly is good test case? There are four attributes that describe the quality of a test case. First, defect detection effectiveness, whether or not it finds defects or at least whether or not it is likely to find defects. An exemplary test case should be more than one thing, thereby reducing the total number of test cases required. The other two attributes are both cost considerations: how economical a test case is to perform, analyze and debug, and how much maintenance effort is required on the test case each time the software changes.

These four attributes must often be balanced one against another. For example, a single test case that tests a lot of things is likely to cost a lot to perform, analyze and debug. It may also require a lot of maintenance each time the software changes. So the skill of testing is not only in ensuring that test cases will find a high proportion of defects, but also ensuring that the test cases are well designed to avoid excessive costs.

Automating tests is also a skill but a very different skill from testing. Many organizations are surprised to find that it is more expensive to automate a test than to perform it once manually. In order to gain benefits from test automation, the tests to be

automated need to be carefully selected and implemented. Automated quality is independent of test quality. (Mark Fewster, 1999)

It doesn't matter how clever you are at automating a test or how well you do it, if the test itself achieves nothing then the end result is a test that achieves nothing faster. Automating a test affects only how economic and evolvable it is. Once implemented, an automated test is generally much more economic, the cost of running it being a mere fraction of the effort to perform it manually. However, automated tests generally cost more to create and maintain. The better approach to automate tests cheaper is to implement them in the long term. If no thought is given to maintenance when tests are automated, updating an entire automated test suite can cost as much as the cost of performing all of the test manually.

For an effective suite of automated test suite you have to start with a good test suite, a set of tests skillfully designed by a tester to exercise the most important thing. You then have to apply automation skills to automate the tests in such a way that they can be created and maintained at a reasonable cost.

4.2 WHAT ARE THE ADVANTAGES OF TEST AUTOMATION

Test automation can enable some testing tasks to be performed far more efficiently than could ever be done by testing manually. There are also other benefits, including those listed below.

- i.* **Fast:** Run more tests more often. A clear benefit of automation is the ability to run more tests in less time and therefore to make it possible to run them more often.
- ii.* **Reusable:** Run existing tests on a new version of a program. This is perhaps the most obvious task, particularly in an environment where many programs are frequently modified. The effort involved in performing a set of regression tests should be minimal.
- iii.* **Repeatable:** Tests that are repeated automatically will be repeated exactly every time. This gives a level of consistency to the tests which is very difficult to achieve manually. The same tests can be executed on different hardware configurations, using different operating systems or using different databases.
- iv.* **Cost Reduction:** As the number of resources for regression test are reduced, skilled testers can put more effort into designing better test cases to be run. There will always be some testing which is the best done manually; the testers can do a better job of manual testing if there are far fewer tests to be run manually.

- v. **Better Quality Software:** Perform tests which would be difficult or impossible to do manually. Attempting to perform a full-scale live test of an online system with say 200 users may be impossible, but the input from 200 users can be simulated using automated tests. By having end users define tests that can be replayed automatically, user scenario tests can be run at any time even by technical staff who do not understand the intricacies of the full business application.
- vi. **Confidence:** Knowing that an extensive set of automated tests has run successfully, there can be greater confidence that there won't be any unpleasant surprises when the system is released.
- vii. **Earlier time to market:** Once a set of tests has been automated, it can be repeated far more quickly than it would be manually, so the testing elapsed time can be shortened.

4.3 COMMON PROBLEMS OF TEST AUTOMATION

There are a number of problems that may be encountered in trying to automate testing. Problems which come as a complete surprise are usually more difficult to deal with, so having some idea of the type of problems you may encounter should help you in implementing your own automation regime. Most problems can be overcome. We describe some of the more common problems below.

- i. **Unrealistic expectations.** Software industry is known for latching onto any new technical solution and thinking it will solve all of our current problems. Testing tools are no exception. There is a tendency to be optimistic about what can be achieved by a new tool. It is human nature to hope that this solution will at last solve all of the problems we are currently experiencing. Vendors naturally emphasize the benefits and successes, and may play down the amount of effort needed to achieve lasting benefits. The effect of optimism and salesmanship together is to encourage unrealistic expectations. If management expectations are unrealistic, then no matter how well the tool is implemented from a technical point of view, it will not meet expectations.
- ii. **Poor testing practice.** If testing practice is poor, with poorly organized tests, little or inconsistent documentation, and tests that are not very good at finding defects, automating testing is not a good idea. It is far better to improve the effectiveness of testing first than to improve the efficiency of poor testing.

- iii. **Expectation that automated tests will find a lot of new defects.** A test is most likely to find a defect the first time it is run. If a test has already run and passed, running the same test again is much less likely to find a new defect, unless the test is exercising code that has been changed or could be affected by a change made in a different part of the software, or is being run in a different environment. Test execution tools are ‘replay’ tools, i.e. regression testing tools. Their use is in repeating tests that have already run. This is a very useful thing to do, but it is not likely to find a large number of new defects, particularly when run in the same hardware and software environment as before. Tests that do not find defects are not worthless, even though good test design should be directed at trying to find defects. Knowing that a set of tests has passed again gives confidence that the software is still working as well as it was before, and that changes elsewhere have not had unforeseen effects.
- iv. **False sense of security.** Just because a test suite runs without finding any defects, it does not mean that there are no defects in the software. The tests may be incomplete, or may contain defects themselves. If the expected outcomes are incorrect, automated tests will simply preserve those defective results indefinitely.
- v. **Maintenance of automated tests.** When software is changed it is often necessary to update some, or even all, of the tests so they can be re-run successfully. This is particularly true for automated tests. Test maintenance effort has been the death of many test automation initiatives. When it takes more effort to update the tests than it would take to re-run those tests manually, test automation will be abandoned.
- vi. **Technical problems.** Commercial test execution tools are software products, sold by vendor companies. As third-party software products, they are not immune from defects or problems of support. It is perhaps a double disappointment to find that a testing tool has not been well tested, but unfortunately, it does happen. Interoperability of the tool with other software, either your own applications or third-party products, can be a serious problem. The technological environment changes so rapidly that it is hard for the vendors to keep up. Many tools have looked ideal on paper, but have simply failed to work in some environments. The commercial test execution tools are large and complex products, and detailed technical knowledge is required in order to gain the best from the tool. Training supplied by the vendor or distributor is essential for all those who will use the tool directly.

In addition to technical problems with the tools themselves, you may experience technical problems with the software you are trying to test. If software is not designed and built with testability in mind, it can be very difficult to test, either manually or automatically. Trying to use tools to test such software is an added complication which can only make test automation even more difficult.

- vii. **Organizational issues.** Automating testing is not a trivial exercise, and it needs to be well supported by management and implemented into the culture of the organization. Time must be allocated for choosing tools, for training, for experimenting and learning what works best, and for promoting tool use within the organization. An automation effort is unlikely to be successful unless there is one person who is the focal point for the use of the tool. Typically, person who is excited about automating testing, and will communicate his or her enthusiasm within the company. This person may be involved in selecting what tool to buy, and will be very active in promoting its use internally. Test automation is an infrastructure issue, not just a project issue. In larger organizations, test automation can rarely be justified on the basis of a single project, since the project will bear all of the start-up costs and teething problems and may reap little of the benefits. If the scope of test automation is only for one project, people will then be assigned to new projects, and the impetus will be lost. Test automation often falls into decay at precisely the time it could provide the most value, i.e. when the software is updated. Standards are needed to insure consistent ways of using the tools throughout the organization. Otherwise every group may develop different approaches to test automation, making it difficult to transfer or share automated tests and testers between groups.

4.3.1 The Limitations of Automating Software Testing

It is not possible to automate all testing activities or all tests. There will always be some testing that is much easier to do manually than automatically, or that is so difficult to automate. Some test should not be automated:

- i. If a test run only once a year, then it is probably not worth automating it.
- ii. If the user interface and functionality change beyond recognition from one version to the next, the effort to change automated tests to correspond is not likely to be cost effective.
- iii. Some test results can be verified by a human, but it is difficult to automate. For example the color of a button can be verified a by manual test but it can be difficult for a automated test.

- iv. If a test involve physical interaction, it is impossible to automate it.

Manual tests find more defects than automated tests. Once an automated test suites has been constructed, it is used to re-run tests. By definition, these tests have been run before and therefore they are much less likely to reveal defects in the software this time.

James Bach reported that in his experience, automated tests found only 15 percent of the defects while manual testing found 85 percent. (Bach, 1997)

Automating a set of tests does not make them any more effective than those same tests run manually. Automation can eventually improve the efficiency of the tests; that is, how much they cost to run and how long they take to run.

Automated tests take more effort to set up than manual tests, because they require effort to maintain.

5. A FRAMEWORK MODEL FOR THE SUCCESS OF THE TEST AUTOMATION

5.1 PROBLEMS WITH TEST AUTOMATION

The common problems are:

- i. Maintenance of the old script when there is a feature change or enhancement.
- ii. The use of the script when we migrate the application from old version to new version.
- iii. The change in technology of the application will affect the old scripts.

Test automation is software development. This principle implies that much of what we know about writing software also applies to test automation. And some of the things we know may not be apparent to people with little or no experience writing software.

Much of the cost of software development is maintenance (changing the software after it is written). This single fact accounts for much of the difference between successful and unsuccessful test automation efforts. Testers in many organizations that attempted test automation only to abandon the effort within a few months. The most common reason is that the tests quickly became brittle and too costly to maintain. The little change in the implementation of the system for example, renaming a button, breaks swarms of tests, and fixing the tests is too time consuming. (Emery, Writing Maintainable Automated, 2009)

But some organizations succeed with test automation though they experience maintenance costs. An important difference is that where unsuccessful organizations are surprised by the maintenance costs, successful organizations expect them. The difference between success and failure is not the maintenance costs, but whether the organization expects them. Successful organizations understand that test automation is software development, that it involves significant maintenance costs, and that they can and must make deliberate, vigilant effort to keep maintenance costs low.

5.2 FROM RECORD/PLAYBACK TO FRAMEWORKS

5.2.1 What Is The Problem With Record/Playback Approach

The test automation tool vendors market their product as the main feature of the tool is the ability to capture the user actions and later to playback them. Here is the basic

paradigm for GUI-based automated regression testing – the so called Record/Playback method (also called as Capture/Replay approach)

- i. Design a test case in the test management tool.
- ii. Using the capture feature of the automation testing tool record the user actions. The result is a macro-like script where each user action is presented.
- iii. Enhance the recorded script with verification points, where some property or data is verified against an existing baseline. Add delay and wait states points where the different actions are synchronized.
- iv. Playback the scripts and observe the results in the log of the test management tool.

The basic drawback in this method is the scripts resulting from this method contain hard-coded values which must change if anything at all changes in our AUT. The costs associated with maintaining such scripts are astronomical, and unacceptable. These scripts are not reliable, even if the application has not changed, and often fail on replay (pop-up windows, messages, and other things can happen that did not happen when the test was recorded).

If the tester makes an error entering data, etc., the test must be re-recorded. If the application changes the test must be re-recorded. All that is being tested are things that already work. Areas that have errors are encountered in the recording process (which is manual testing, after all). These bugs are reported, but a script cannot be recorded until the software is corrected. So logically nothing is tested by this approach.

This method is fraught with problems, and is the most costly (time consuming) of all methods over the long term. The record/playback feature of the test tool is useful for determining how the tool is trying to process or interact with the application under test, and can give us some ideas about how to develop your test scripts, but beyond that, its usefulness ends quickly.

5.2.2 Types Of Test Automation Frameworks

5.2.2.1 Data Driven Test Frameworks

In Data-Driven Testing the input and output values for the test are fetched from data files. These data files can be excel files, CVS files, ADO objects, DAO objects or any ODBC source etc. Subsequently these get loaded into various variables in scripts which might be either manually created or can be captured ones. For input values as well as output verification values, variables are used in data-driven

framework. The test scripts contain all the coded information regarding reading of the data files, navigation through the application & test status logs etc.

5.2.2.1.1 Advantage of Data Driven Automation Test Framework

- i. We can create our scripts even when development of application is still going on.
- ii. Redundancy & unnecessary duplication of creation of automated testing scripts gets greatly reduced due to the modular type of design due to the use of files or records for both input as well as verifying the data.
- iii. In case of any change in functionality, we just need to revise the particular "Business Function" script.
- iv. Information like data inputs or outputs, expected results get stored in the form of conveniently managed text records.
- v. This permits better error handling; thereby the resulting test scripts are more robust. This is due to the fact that when a script is called, the functions return "TRUE" or "FALSE" values, instead of aborting.

5.2.2.2 Keyword Driven Test Frameworks

This requires the development of data tables and keywords, independent of the test automation tool used to execute them and the test script code that "drives" the application-under-test or AUT and the data. Keyword-driven tests look very similar to manual test cases. In a keyword-driven test, the functionality of the application-under-test is documented in a table as well as in step-by-step instructions for each test. In this method, the entire process is data-driven, including functionality.

5.2.2.2.1 Advantages of Keyword Driven Framework

- i. We can write the detailed test plan having desired inputs and verification data in the form of simple spreadsheets.
- ii. In case some expert having expertise of scripting language used by the automation tool, can create utility scripts well before the creation of the detailed test plan, then the testing engineer is able to use the automation tool immediately through the spreadsheet input method. For this he/she need not master the concerned scripting language actually used by the automation tool.

- iii. The testing engineer becomes productive with the new testing tool far quickly, since he/she is required to understand the particular format for use in the test plan & various desired keywords.

5.2.2.3 Hybrid Test Frameworks

The most commonly implemented framework is a best combination of all the techniques. It combines keyword-driven, library and data-driven frameworks. Hybrid Testing Framework allows data driven scripts to take advantage of the powerful libraries and utilities that usually accompany a keyword driven architecture. The framework utilities can make the data driven scripts more compact and less prone to failure. Tests are fully scripted in a Hybrid Testing Framework thus increasing the automation effort. Hybrid Testing Framework also implements extensive error and unexpected windows handling. It is used for automation of medium to large applications with long shelf life.

5.3 CLASS-DRIVEN FRAMEWORK APPROACH

The class based approach aims to implement a framework that is based on the mapping of UI object to the test class. Test class can be a window, a modul or a web page. If AUT has five diffirent pages than we will define five test automation class.

6. CLASS-DRIVEN MODEL FOR MAINTAINABLE TEST AUTOMATION

Class-driven model aims to map GUI object to a test class. Test class will be structured according to the functionality of the GUI object. But it doesn't mean to map each object to a unique class. There will be main classes that includes many GUI objects. And each object should be a member of a class.

Class-driven testing enables automated test development in a modular, object-oriented manner. Each automated test case is designed as a testing class that owns test data and knows how to manipulate it. Testers can create multiple instances of each testing class and populate them with different data to increase the automated test coverage.

With class-driven testing, the entire test automation framework can be designed using nothing but testing classes and objects, thus making test projects more flexible and easily maintainable than ordinary automated tests.

Class-driven model aims to reduce the maintenance cost of the test automation project. Thus, time to market will be reduce.

6.1 CONCEPT OF CLASS IN GENERAL

In object-oriented programming, a class is a construct that is used as a blueprint to create instances of itself. A class defines constituent members which enable these class instances to have state and behavior. Data field members (member variables or instance variables) enable a class object to maintain state. Other kinds of members, especially methods, enable a class object's behavior. Class instances are of the type of the associated class.

Classes can be derived from one or more existing classes, thereby establishing a hierarchical relationship between the derived-from classes (base classes or parent classes or superclasses) and the derived class (or child class or subclass) . The relationship of the derived class to the derived-from classes is commonly known as an is-a relationship. For example, a class 'Button' could be derived from a class 'Control'. Therefore, a Button is a Control. Structural and behavioral members of the parent classes are inherited by the child class. Derived classes can define additional structural members (data fields) and/or behavioral members (methods) in addition to those that they inherit and are therefore specializations of their superclasses. Also, derived classes can override inherited methods if the language provides for such.

Conceptually, a superclass should be considered as a common part of its subclasses. This factoring of commonality is one mechanism for providing reuse. Common

attributes are derived from the base class. It provides great convenience on change management. I design class-driven test automation model according to this concept. Since test automation is also software development, we have to thought about change management.

6.2 ADVANTAGES OF OBJECT ORIENTED BASIC CONCEPT

These are some of the major advantages of OOP.

Simplicity: Software objects model real world objects, so the complexity is reduced and the program structure is very clear.

Modularity: Each object forms a separate entity whose internal workings are decoupled from other parts of the system.

Modifiability: It is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.

Extensibility: Adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones.

Maintainability: Objects can be maintained separately, locating and fixing problems easier.

Re-usability: Objects can be reused in different programs.

Class-driven test automation is designed over these OOP advantages. Maintainability and extensibility of the test automation is very important in software project.

6.3 WHAT DOES TEST CLASS MEAN IN CLASS-DRIVEN MODEL

Test class is not identical with class concept in object-oriented approach. But they are essentially similar. Common attributes of test objects can be defined on a class. Base classes defines most common attributes and subclasses defines special attributes of each GUI objects .

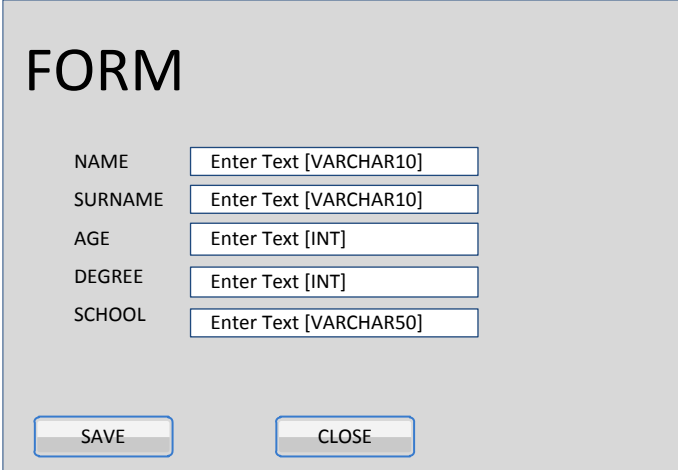
Class-oriented architecture associate each GUI object to a test class. As some of the GUI objects correspond to a test class directly, some GUI objects will be only an attribute of a test class.

There is no inheritance between classes. Each class has its own properties and methods.

6.4 THE BENEFIT OF THE MODEL

According to class-driven theory, each unique GUI or Application object will be created as a class instance. It provides us to manage object properties and methods from class. And test automation scripts are written by using objects. Since objects' properties are managed from the class, any change on the object is triggered as an update on the class. Since object properties is defined on the class, it does not require changes on the scripts. An object is used in multiple test script so a single change in test class will be sufficient. Testers don't need to update all test scripts individually.

Figure 6.1: An Application Form



The image shows a screenshot of a graphical user interface window titled "FORM". The window has a light gray background and a thin border. At the top left, the word "FORM" is displayed in a large, bold, black font. Below the title, there are five input fields arranged vertically, each with a label to its left and a text box to its right. The labels and their corresponding text boxes are: "NAME" with "Enter Text [VARCHAR10]", "SURNAME" with "Enter Text [VARCHAR10]", "AGE" with "Enter Text [INT]", "DEGREE" with "Enter Text [INT]", and "SCHOOL" with "Enter Text [VARCHAR50]". At the bottom of the window, there are two buttons: "SAVE" on the left and "CLOSE" on the right. Both buttons have a light gray background and a thin border.

In Figure 4-1, there is an application window named FORM. This form has five input fields and two buttons. User fills all fields and clicks save. For testing this form we should fill all the fields according to the some combinations. For example first combination can be like this:

Test Script 1:

Step 1: Enter a text with 10 characters in NAME field.

Step 2: Enter a text with 10 characters in SURNAME field.

Step 3: Enter an integer in AGE field.

Step 4: Enter an integer in DEGREE field.

Step 5: Enter a text with 50 characters in SCHOOL field.

Step 6: Click SAVE button.

Test Script 2:

Step 1: Enter a text with more than 10 characters in NAME field.

Step 2: Enter a text with 10 characters in SURNAME field.

Step 3: Enter an integer in AGE field.

Step 4: Enter an integer in DEGREE field.

Step 5: Enter a text with 50 characters in SCHOOL field.

Step 6: Click SAVE button.

Test Script 3:

Step 1: Enter a text with 10 characters in NAME field.

Step 2: Enter a text with more than 10 characters in SURNAME field.

Step 3: Enter an integer in AGE field.

Step 4: Enter an integer in DEGREE field.

Step 5: Enter a text with 20 characters in SCHOOL field.

Step 6: Click SAVE button.

Test Script 4:

Step 1: Enter a text with 10 characters in NAME field.

Step 2: Enter a text with 10 characters in SURNAME field.

Step 3: Leave blank AGE field.

Step 4: Enter an integer in DEGREE field.

Step 5: Enter a text with 50 characters in SCHOOL field.

Step 6: Click SAVE button.

Number of test scripts can be increased in parallel to combinations. Assume that 10 different test scripts are written for testing this form and added to regression set. In Record/Playback systems, each test script is recorded and saved as an automated test script. And these 10 scripts are run (playback) in each release. This system works successfully in practice until the form changes. For example, the name of the SURNAME field changes as SNAME. After this tiny change on the form, if automated scripts run, none of them completed successfully since they couldn't find SURNAME field. These 10 scripts should be recorded again.

In class-driven model, FORM is defined as a test class. Five fields are the attributes of the class and two buttons can be defined as functions in the class definition. Test scripts are written by using FORM class.

Class form

Attribute 1 :text field name (NAME), varchar(10),set(),get()

Attribute 2 :text field name (SURNAME), varchar(10),set(),get()

Attribute 3 :text field name (AGE), int, ,set(),get()

Attribute 4 :text field name (DEGREE), int,set(),get()

Attribute 5 :text field name (SCHOOL), varchar(50) ,set(),get()

Attribute 6 :click(),button name(SAVE)

Attribute 7 :click(),button name(CLOSE)

End Class

Test Script 1:

Step 1: form.Attribute1.set("aaaaaaa")

Step 2: form.Attribut2.set("xxxxxxxxx")

Step 3: form.Attribute3.set("54")

Step 4: form.Attribute4.set("85")

Step 5: form.Attribute5.set("xxxx")

Step 6: form.Attribute6.click()

If the name of the SURNAME field changes as SNAME, for updating test script, only changing the name of the attribute from the test class is enough.

Class form

.....

Attribute 2 :text field name (SNAME), varchar(10),set(),get()

.....

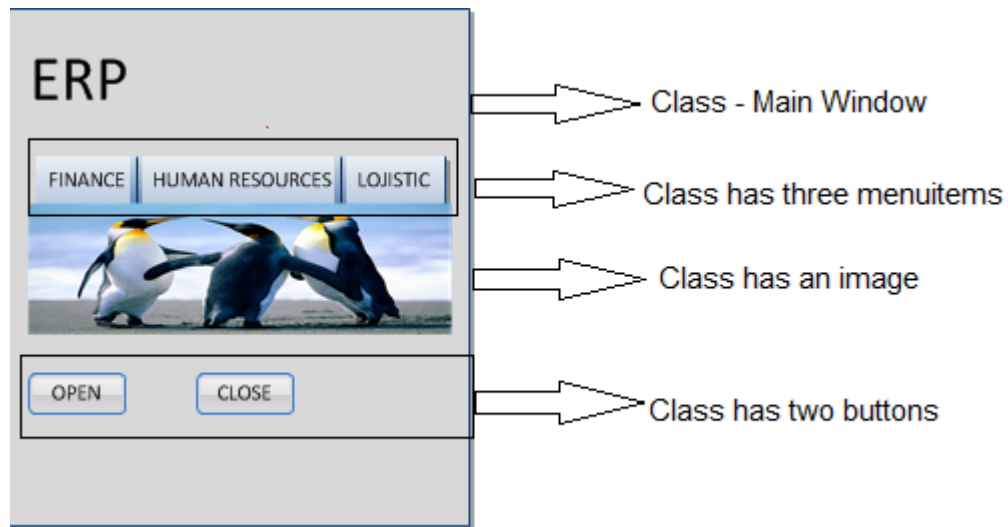
End Class

In class-driven approach, each object should be defined on a class even as a function or property. In this way, everything on the GUI or application is considered as an object and defined as an attribute of test-class. So on the test scripts, instances of test classes are created and by sending any parameters function of attributes are reached.

6.5 STRUCTURE OF CLASS-DRIVEN APPROACH

Window-based applications have pages,moduls,menus, buttons,etc. For opening the application, we should run the execution file of application. Then application is opened. Generally each application has a main page. Main page or window have some menus, fields or buttons. In class-driven model, this main window is a class and menus, buttons and other GUI objects on the window are the structural or behavioral properties of the main class.

Figure 6.2: Main Page of Application



In class-driven model, main page is a class that has three menuItems, two buttons, one image and a title. All these GUI objects are attributes of the class.

Class ERP

Attribute FINANCE - menuItem

Attribute HUMAN RESOURCES- menuItem

Attribute LOJISTICS- menuItem

Attribute IMAGE

Attribute TITLE

Attribute OPEN

Attribute CLOSE

End Class

Here all these attributes have click event. So each can be defined as a function.

```
f_select_finance ()
```

```
    {Window(ERP).menuItem(FINANCE).click}
```

```
f_select_hr ()
```

```
    {Window(ERP).menuItem(HUMAN RESOURCES).click}
```

```
f_select_logistics ()
```

```
    {Window(ERP).menuItem(LOJISTICS).click}
```

```
f_open ()
```

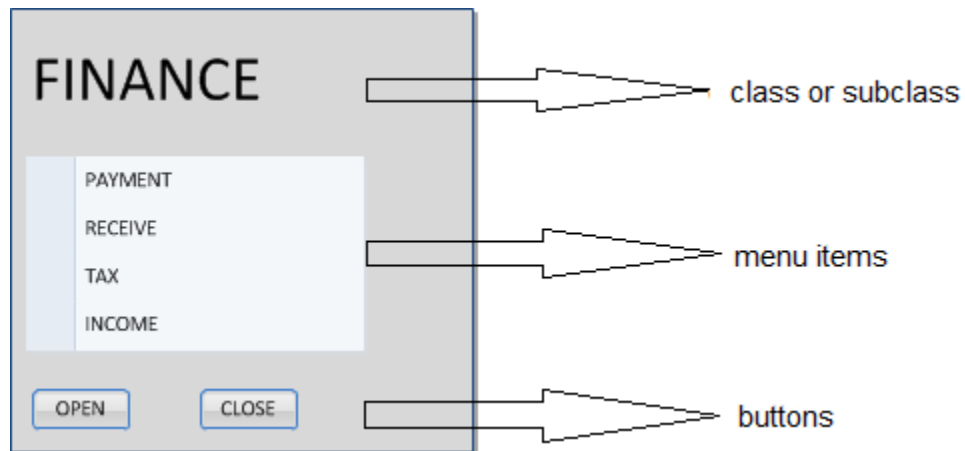
```
    {Window(ERP).button(OPEN).click}
```

```
f_close ()
```

```
    {Window(ERP).button(CLOSE).click}
```

When writing test script, create an instance of class ERP and use its attributes or methods.

Figure 6.3: Finance Modul in ERP



Finance modul has menu that includes four menu items and two buttons. So Finance modul is described as a test class.

Class finance

Attribute menu

Attribute OPEN

Attribute CLOSE

End Class

Here all these attributes have click event. So each can be defined as a function.

```
f_select_menuItem (PAYMENT)
```

```
{ Modul(FINANCE).menuItem(PAYMENT).click }
```

```
f_open ()
```

```
{ Modul(FINANCE).button(OPEN).click }
```

```
f_close ()
```

```
{ Modul(FINANCE).button(CLOSE).click }
```

Figure 6.4: Payment Page

PAYMENT

Name: Currency:

Surname: Date:

Adress: Type:

Tel: Status: InProgress

Urgent Suspended

Pay Waiting

Annotations:

- subclass - payment
- textfiles, comboboxes, checkboxes, radiobuttons
- buttons

Payment is page that includes five text fields, three comboboxes, two checkboxes and five buttons. So payment page is described as a test class. We can also think this page as a subclass of FINANCE class. But the main idea here is the accessing PAYMENT page over the FINANCE modul. There is no inheritance between the instances of finance.class and payment.class. It is necessary to open FINANCE first then it is possible to open PAYMENT.

Class payment

Attribute name

f_set_name()

f_get_name()

.....

Attribute surname

f_set_surname()

f_get_surname()

.....

Attribute address

.....

Attribute tel

.....

Attribute currency

.....

Attribute date

.....

Attribute date

.....

Attribute type

.....

Attribute total

.....

Attribute urgent

.....

Attribute pay

.....

Attribute open

.....

Attribute reject

.....

Attribute cancel

.....

End Class

In this part, according to a basic test scenario, test automation script is written according to the class-driven approach.

Test scenario: According to the information entered in the payment form, submission is done successfully.

Test scenario is converted to the test steps:

1. Open ERP application.
2. Open Finance modul.
3. Open Payment section.
4. Fill all the fields on payment form.
5. Click Submit.

Step 1 : Open ERP application.

Test Script: f_open_erp();

Step 2 : Open Finance Modul.

Test Script: erp. f_select_finance ();

erp.f_open();

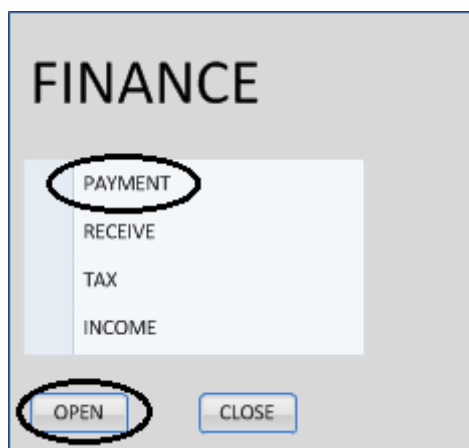
Figure 6.5: ERP Form



Step 3 : Open Payment Section.

Test Script : `finance.f_select_menuItem(PAYMENT);`
`finance.f_open();`

Figure 6.6: Finance Modul



Step 4 : Fill Payment Form.

Test Script: payment.f_set_name(SERKAN);
payment.f_set_surname(AKOGLANOGLU);
payment.f_set_adress(ISTANBUL);
payment.f_set_tel(530425035);
payment.f_set_currency(TL);
payment.f_set_date(01/01/2012);
payment.f_set_type(PEŞİN);
payment.f_set_total(500);
payment.f_submit();

Figure 6.7: Payment Section

PAYMENT

| | | | |
|----------|--|-----------|---|
| Name: | <input type="text" value="SERKAN"/> | Currency: | <input type="text" value="TL"/> |
| Surname: | <input type="text" value="AKOGLANOGLU"/> | Date: | <input type="text" value="01/01/2012"/> |
| Adress: | <input type="text" value="ISTANBUL"/> | Type: | <input type="text" value="PEŞİN"/> |
| Tel: | <input type="text" value="5304556759"/> | Total: | <input type="text" value="500"/> |
| | <input checked="" type="checkbox"/> Urgent | | |
| | <input checked="" type="checkbox"/> Pay | | |

Test script is ready for run. In practice, if there won't be any changes on the application under test, then it is not necessary to write test scripts by using any methodology. Already for this kind of software does not make sense to write test automation. So in general, test automation is written for the applications that are continued to the development and changes.

6.6 HOW TO HANDLE CHANGES BY CLASS-DRIVEN APPROACH

If any change made in the application requires an update on the test scenario, then it requires any change on the test automation script. The most important benefit of test automation is it is faster than manual test. But it takes a lot of time for maintenance or modification than estimated by record/playback methods. Due to increased cost of test automation maintenance over time more than expected, test automation project fails.

To avoid maintenance costs more than expected, an test automation architecture should be identified that takes into account of maintenance costs. Class-driven architecture aims to write most maintainable test automation scripts.

Figure 6.8: Change on the Payment

The screenshot shows a web form titled "PAYMENT". The form contains the following fields and controls:

- Name:
- Surname:
- Address:
- Tel:
- Currency:
- Date:
- Type:
- Status: A dropdown menu with three options: InProgress, Suspended, and Waiting. This section is highlighted with a black border.
- Urgent:
- Pay:

At the bottom of the form, there are five buttons: SUBMIT, REJECT, CANCEL, SUSPEND, and CLOSE.

Change made on the form shown above. A new field, Status, is added to form instead of field Total. Third step of the test case is changed. So test script have to be updated simultaneously.

1. Open ERP application.
2. Open Finance modul.
- 3. Open Payment section.**
4. Fill all the fields on payment form.
5. Click Submit.

Step : Fill Payment Form.

Test Script: payment.f_set_name(SERKAN);
 payment.f_set_surname(AKOGLANOGLU);
 payment.f_set_adress(ISTANBUL);
 payment.f_set_tel(530425035);
 payment.f_set_currency(TL);

```
payment.f_set_date(01/01/2012);  
payment.f_set_type(PEŞİN);  
payment.f_set_total(500);  
payment.f_set_status(Waiting);  
payment.f_submit();
```

Just changing one line of code is enough.

```
payment.f_set_total(500) → payment.f_set_status(Waiting)
```

No need to re-record the same scenario. Adding a new feature to the class is enough. If there are more than one scenario that uses the same screen, the only change made in one place will be sufficient.

7. A CASE STUDY

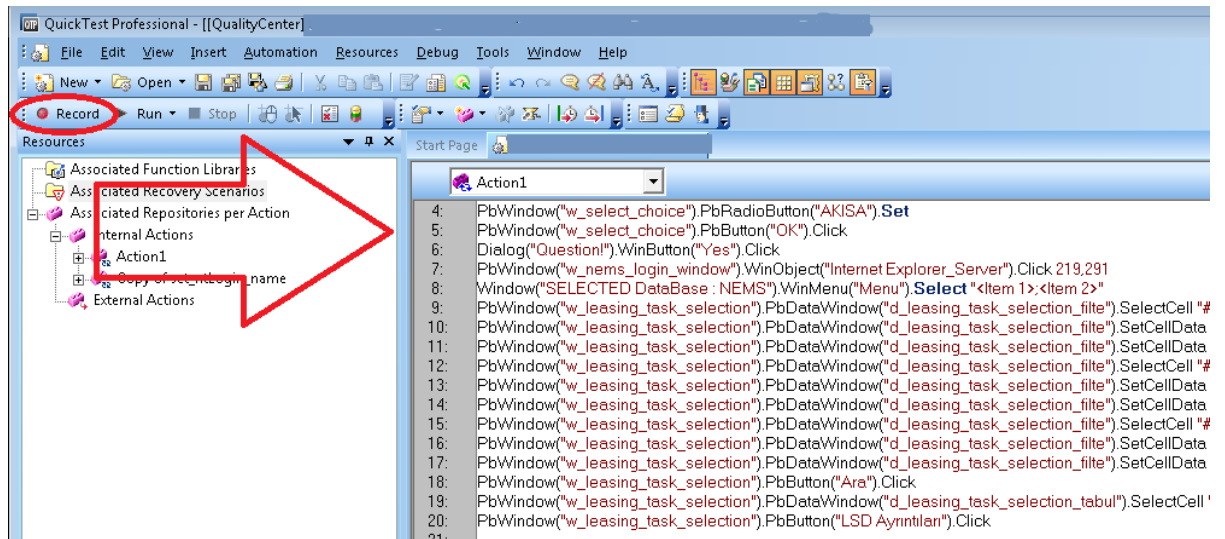
I am a software test architect at a telecommunication company. I am responsible for the overall quality of a modul in a software system that is using for network management system. It is written with PowerBuilder 12.0 by Sybase. It is a window-based, client server application.

I am responsible for a modul, Leasing Survey Management, that provides users to handle the leasing management of the GSM base station. The system works briefly as follows. A renting planner decides to rent a building or tower and creates a work order form on the system. The work order flows through the steps of it. In each step, different users login and input any data or information and another user submit it. Each user login with different user type that has different user rights. Approximately 100 users use this leasing modul. Leasing modul also has integration with two different moduls. Integration with other systems is provided by web services. Two hundred and fifty different business requirements are defined by product owners. This means we have at least two hundred and fifty test scenarios.

We release a new version of the system each week. Before the relase we test the new features and then run automated test scripts. Before running the automation, we should modify the automated test scripts according to the changes on the system or changes on the business requirement.

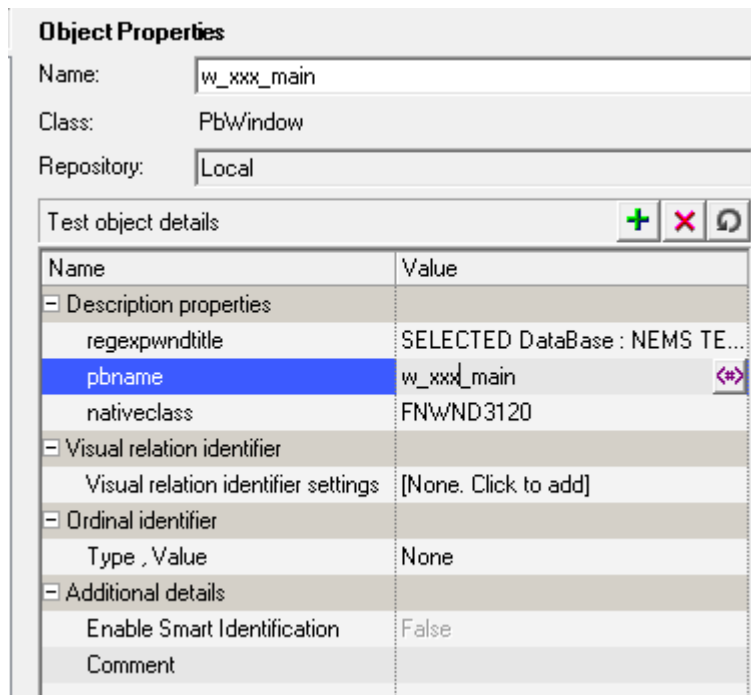
We are using HP QTP version 11.0 for test automation. It has record/playback or capturing feature. We record the user actions according to the requirements and save it as automated test script. QTP converts the recorded user actions to VB Script. Then we playback each automated test.

Figure 7.1: QTP Recording and Converting Screen



When we click the Record button on the QTP main page, it begins to capture all user actions on the screen and converts each action to a line of VB Script. In the background it generates an object repository that holds the objects details. In the script above, QTP creates it and saves as a test. Then tester run it whenever it is needed.

Figure 7.2: QTP Object Repository

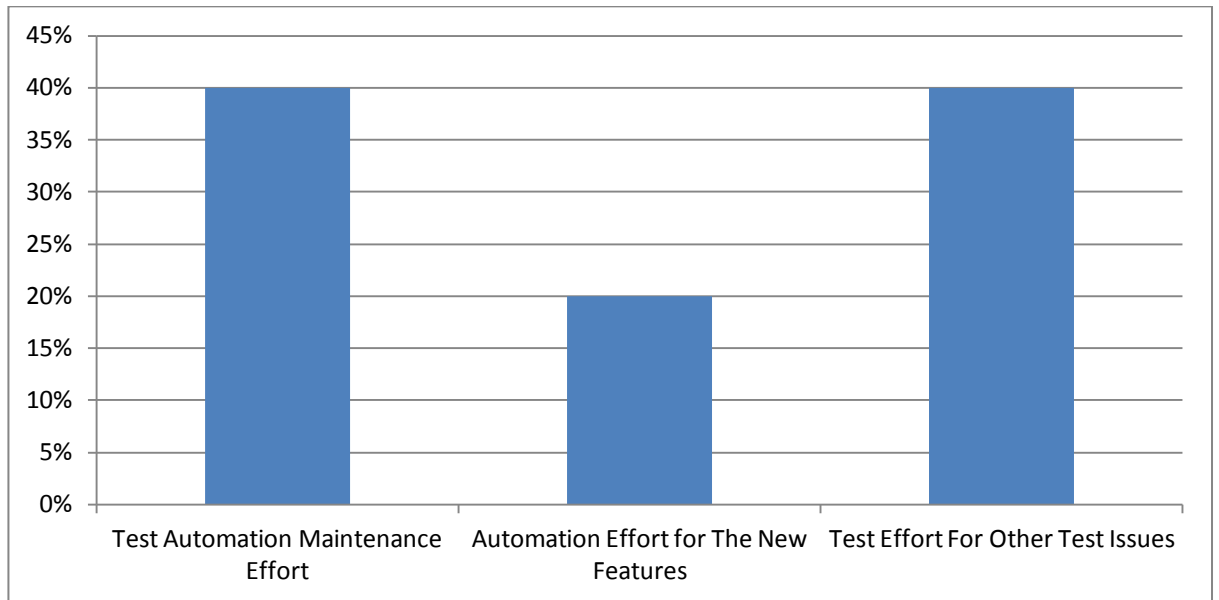


Object repository keeps the own properties of the each object. It is also generated by QTP automatically. Details of each object can be handled from here.

The problem in this methodology is the difficulty of modifying the recorded test scripts. Any minor change on the main window requires some update on the test scripts that uses this window. Hence we have to allocate time for the maintenance of the older test scripts.

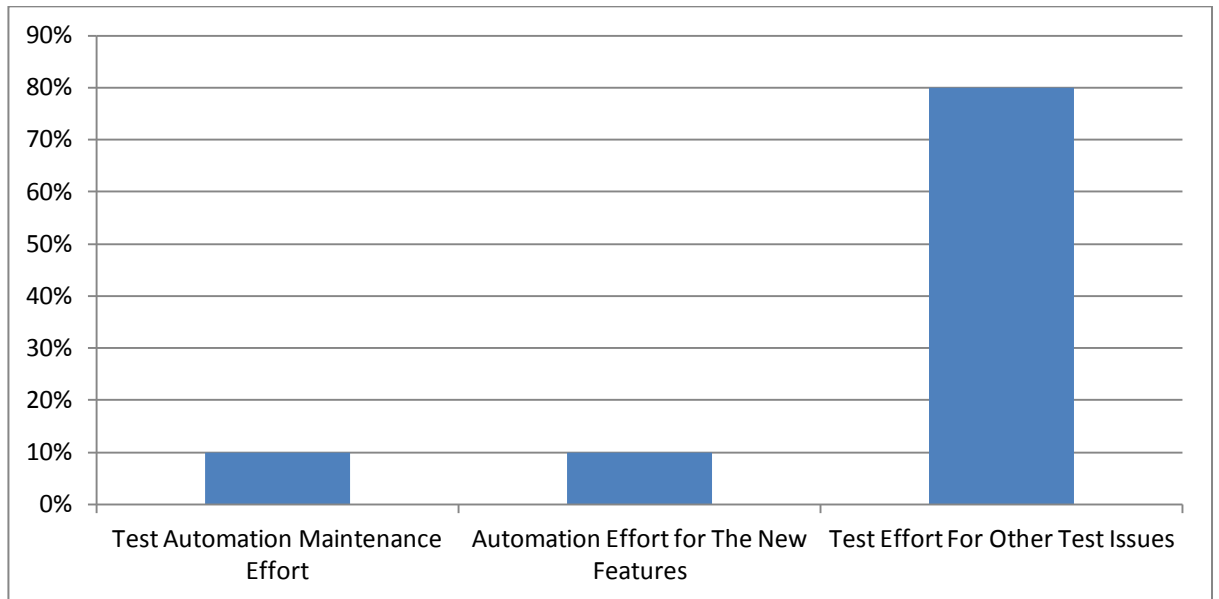
For leasing modul, each week I give two and a half day for all testing process. One day for the maintenance of the recorded test scripts. But it is very long time for the overall test time. 40 percent of the test effort for maintenance is not acceptable for the project management. I give 20 percent of the test effort for the automation of new features. And 40 percent for the other test issues.

Figure 7.3: Test Automation Maintenance Effort With Classic Method



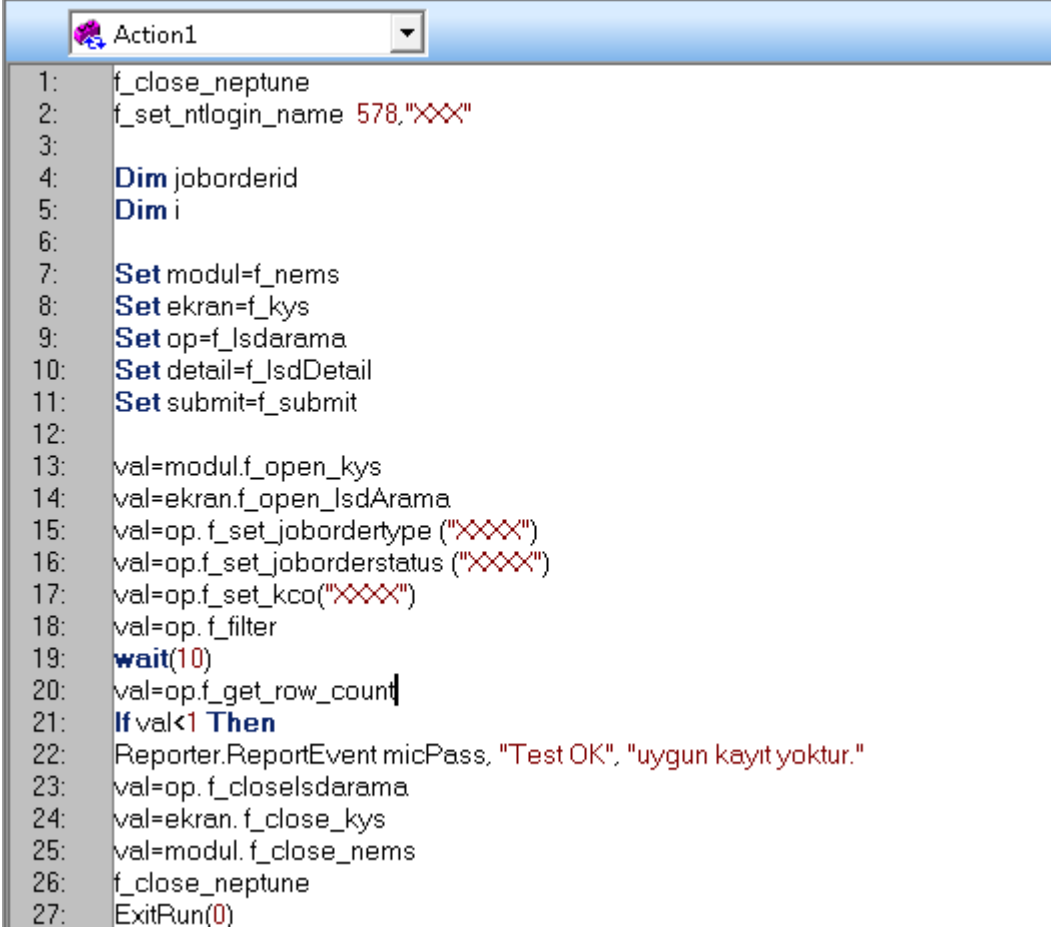
Every release, approximately 8 requirements are changed and 2 new requirements are added for Leasing module. Figure 7-3 shows the test automation effort with classic test automation method for each release.

Figure 7.4: Test Automation Maintenance Effort With Class-Driven Architecture



By using the class-driven architecture, not only the time spent for maintenance but also decrease the time spent for new test scripts. Figure 7-4 shows the test effort distribution with class-driven approach.

Figure 7.5: A Test Script written with Class-Driven



```
1: f_close_neptune
2: f_set_ntlogin_name 578,"XXX"
3:
4: Dim joborderid
5: Dim i
6:
7: Set modul=f_nems
8: Set ekran=f_kys
9: Set op=f_lsdarama
10: Set detail=f_lsdDetail
11: Set submit=f_submit
12:
13: val=modul.f_open_kys
14: val=ekran.f_open_lsdArama
15: val=op.f_set_jobordertype ("XXX")
16: val=op.f_set_joborderstatus ("XXX")
17: val=op.f_set_kco("XXX")
18: val=op.f_filter
19: wait(10)
20: val=op.f_get_row_count
21: If val<1 Then
22: Reporter.ReportEvent micPass, "Test OK", "uygun kayıt yoktur."
23: val=op.f_closetarama
24: val=ekran.f_close_kys
25: val=modul.f_close_nems
26: f_close_neptune
27: ExitRun(0)
```

In the figure 7-5, there is a test script that is written by using functions and classes.

“f_set_ntlogin_name” is a function that changes the user type by given parameter. QTP creates the class instances by using “Set” keyword. A function for calling the class should be created in the class definition.

“Set modul=f_nems” means an instance of class “modul” is created. Thus, we can reach properties and methods of class modul from the test script by instances.

Figure 7.6: A Test Class

```
1: Option Explicit
2:
3: public Function f_nems
4: Set f_nems=New nems
5: End Function
6:
7: Class nems
8: 'open neptune
9: Public function f_open_neptune
10: PbWindow("w_nems_login_window").WinObject("Internet Explorer_Server").Click 210,209
11: End Function
12:
13: 'close neptune
14: Public function f_close_neptune
15: PbWindow("w_cp_main").Close
16: End Function
17:
18: 'open kys
19: Public function f_open_kys
20: PbWindow("w_nems_login_window").WinObject("Internet Explorer_Server").Click 215,278
21: End Function
22:
23: 'close kys
24: Public function f_close_kys
25: PbWindow("w_lsd_main").Close
26: End Function
27:
```

Figure 7-6 shows the definition of a class. We should create a function for accessing the class definition from test script. “f_nems” is a function that provides access to this class.

In this architecture, test scripts are written by using class instances. On the test scripts we call functions by sending parameters. All user’ inputs from the GUI are set as parameter from the test script. No special code is written on the test script. Any change or update on the GUI should be handled from the class.

Folder structure is another important area. Class files and test scripts should be saved differently.

❖ **Classes**

- ClsFile
 - Methods
 - Attributes
- ClsNotepad
- ...
- ClsOperations
 - ID
 - Menupath
 - Values

❖ **Test Cases**

- TestCase1
- TestCase2
- TestCase2

8. CONCLUSION AND FUTURE WORKS

Software testing is an important stage in the software projects lifecycles. It is one of the most expensive stages. Effective testing automation is expected to reduce the cost of testing. GUI is increasingly taking a larger portion of the overall program' size and its testing is taking a major rule in the whole project's validation. GUI test automation is a major challenge for test automation. Most of the current GUI test automation tools are partially automated and require the involvement of the users or testers in several stages.

There is a great need in the software test automation area to use an architecture to create automated test scripts on the large scale test automation projects.

Furthermore, there is a need in the test automation area to standardize the concepts and metrics. Design patterns should be generated like software design patterns.

A set of guidelines like coding standards , test-data handling , object repository treatment etc... which when followed during automation scripting produce beneficial outcomes like increase code re-usage , higher portability , reduced script maintenance cost etc. These are just guidelines and not rules; they are not mandatory and you can still script without following the guidelines. But you will miss out on the advantages of having a framework.

The applications of class-driven architecture can help to handle test automation projects from begining to the end. Analysis of GUI objects within the class-driven framework will ensure maintainability of test automation. Changes on the GUI can be handled from the test class of related GUI object on test automation. Thus, not only minor changes but also major version updates can be handled. Once you change the class definiton, there is no need to change test scripts.

REFERENCES

Books

Kaner C., Bach J. B.(2002) *Kaner, Bach and Pettichord's book "Lessons Learned in Software Testing"*.

Emery, D. H. (2009). *Writing Maintainable Automated. Agile Testing Workshop at Agile Development* , 5.

Bach, J. (1997). *Software Test Automation Snake Oil* .

Garrett, T. *Useful Automated Software Testing Metrics By Thom Garrett IDT, LLC*.

Gear, L. (2004). *Achieving the Full Potential of Software Test Automation*

Mark Fewster, D. G. (1999). *Software Test Automation*. Great Britain: ACM Press.

Black R. (2002). *Fundamentals of Software Testing*

Beizer B(1990). *Software Testing Techniques*, Second Edition

Periodicals

Getting Automated Testing Under Control, 1. S. (1999). Getting Automated Testing Under Control, 1999, STQE Magazine.

Kaner, C. Improving the Maintainability of Automated Test Suites, Cem Kaner. Paper Presented at Quality Week '97

Other Resources

Harshada Kekare. (2011). buzzle.com. buzzle.com: <http://www.buzzle.com>

Hoffman, D. (tarih yok). The papers of Doug Hoffman on test automation. The papers of Doug Hoffman on test automation: <http://www.softwarequalitymethods.com>

TestingPerspective: <http://www.testingperspective.com/tpwiki>

Kaner, C. Kaner's Black Box Software Testing Course.

Kohl, J. Jon Kohl's work on "Man and Machine", or the cyborg approach (instead of computer-alone test execution and evaluation).

Parker, K. (APM tools: Applying automated testing earlier in the development lifecycle, Kevin Parker).

TestingStuff: www.testingstuff.com/autotest.html

Emery, D. H. (2009). Writing Maintainable Automated Acceptance Tests.

Bach, J. James Bach's work on /cognitive/ testing.