Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2018

Luís Filipe
da Silva Teixeira

API para procedimentos de teste em sistemas embutidos

A frontend API for test procedures in embedded systems

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações, realizada sob a orientação científica do Doutor João Paulo Silva Barraca, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

agradecimentos

À minha mãe, pelo esforço e apoio que tornou possível alcançar este objetivo.

Ao meu orientador, professor doutor João Paulo Barraca, pela disponibilidade e essencial contributo para este trabalho.

A todos os meu colegas da Critical Software, em especial, ao João Esteves, Tiago Rodrigues e Hugo Almeida por todo o apoio e conhecimento passado sempre com atitude profissional e amiga.

À minha namorada Adriana, pelo apoio e por muitas vezes “aturar” a minha frustração e má disposição nos dias menos bons durante esta caminhada.

palavras-chave

Procedimentos de teste, aplicação de controlo de satélite, simulador.

resumo

O caso de estudo proposto toma uma aplicação de controlo de um satélite como um sistema a ser testado e implica a extensão de um simulador simples para o componente do sistema com o qual esta aplicação interage. Em cima destes devem ser desenvolvidos procedimentos de teste para estudar se há limitações práticas ao usar procedimentos simples para testar interações complexas entre a aplicação de controlo do satélite e o ambiente simulado.
O objetivo do trabalho proposto é definir uma API de teste que possibilite procedimentos de teste simples e forneça todos os meios necessários para testar interações complexas máquina a máquina, tendo o sistema sob teste características de tempo real.

keywords

Test procedures, satellite control application, simulator.

abstract

The proposed case study takes a satellite control application as system under test and entails extending a simple simulator for the system component this application interacts with. Atop of these shall be developed test procedures to study whether there are practical limitations of using simple procedures to test complex interactions between the satellite control application and its simulated environment.
The objective of the proposed work is to define a test frontend API that enables simple test procedures while providing all the means required to test complex machine-to-machine (M2M) interactions, having the system under test (SUT) hard real-time characteristics.

Table of contents

Tables Index

Figures Index

Acronyms

AOCS	Attitude and Orbital Control System
API	Application Programming Interface
BC	Bus Controller
CPU	Central Processing Unit
DBP	Data Block Protocol
ESA	European Space Agency
FEE	Frontend Electronics
GNSS	Global Navigation Satellite System
HITL	Hardware In The Loop
HP	Hewlett Packard
HW	Hardware
M2M	Machine-to-Machine
MSI	Multispectral Instrument
OBC	Onboard Computer
OBSW	On-Board Software
OOP	Object-oriented Programming
PCDU	Power Conditioning and Distribution Unit
QA	Quality Assurance
QTP	Quick Test Pro
RIU	Remote Interface Unit
RO	Radio Occultation
RT	Remote Terminal
SDLC	Software Development Lyfe-Cycle
SQA	Software Quality Assurance
SUT	System Under Test
SVF	Software Verification Facility
SVTS	System Verification Test Specification
SW	Software
TC	Telecommand
TM	Telemetry
UFT	Unified Functional Testing
VCU	Video Compression Unit

1 Introduction

1.1 *Context*

Who polices the police? This is a question many educated people have already made. For the countries that conform to the code of law, the answer lies on the checks and balances that are inscribed in the law and govern the separation of powers of the branches of state. From a more philosophical but also practical perspective the question one is looking to answer is how to assert something is correct without entering and infinite regression of checks?

The question "who polices the police?" finds also key relevance in the field of software, embedded systems and cyber-physical systems testing. These days, testing, as performed in efficient organisations, is mainly automated. Manual testing still exists, because some test procedures are hard, if not impossible to automate in a practical way, but for the most part tests are implemented by automated test procedures. But automated test procedures, at least those that matter for this discussion, are not generated automatically, they are coded by someone. The question is then, if automated test procedures are used to test the correct functioning of a software (SW) application or of a system, are they less prone to errors or somehow more reliable than the SW or system under test? The short answer is "no"; in general, there is no good justification to claim that the source code of an automated test procedure is more reliable than the SW being tested. Rather often one could claim the exact opposite instead, at least for developers own unit tests.

The case that we build here is that, in general, source code of test procedures can be no more error prone than the source code of the SW application being tested. Or better saying, for source code of equivalent complexity one should expect an equivalent error rate. And from here we reach to the following key challenge that we formulate out of first-hand experience: testing complex systems often involves test scripts nearly as complex, or sometimes more complex, than the system being tested.

A potential way around this conundrum is to develop test procedures on top of simple test Application Programming Interfaces (APIs), strictly enforce complexity limitation conventions, use simple special purpose test languages, or a combination of these. The question that we set at the core of this thesis proposal, is whether there is a practical way to test complex systems without having complex test procedures and without creating the illusion of simplicity by hiding complexity in draconian test libraries.

1.2 *Objectives*

The objective of the proposed work is to define a test frontend API that enables simple test procedures while providing all the means required to test complex machine-to-machine (M2M) interactions, having the system under test (SUT) hard real-time characteristics. The case study supporting the proposed work is a satellite instrument control application - the high-level illustration of the test environment is illustrated in Figure 1.

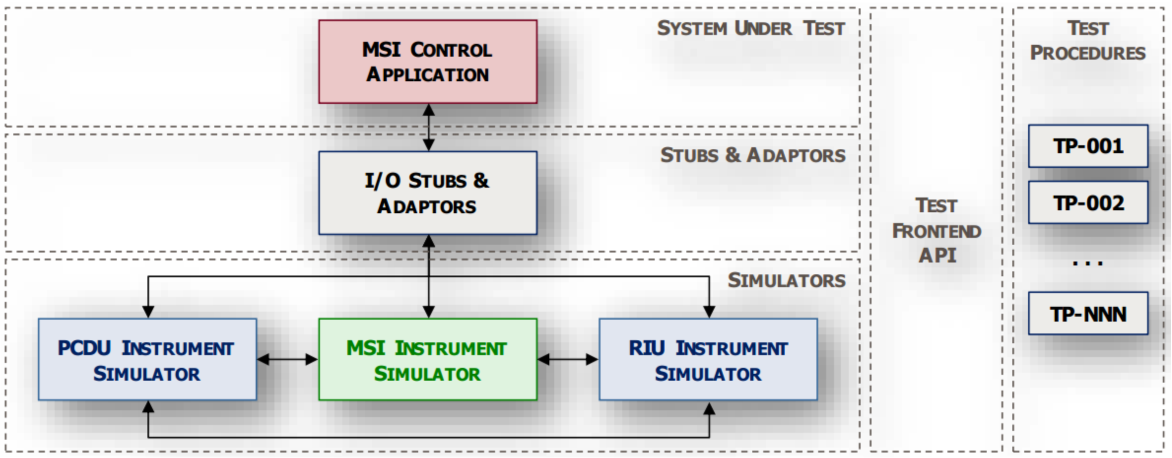


Figure 1 - Proposed Case-Study: Satellite Payload Control Application

The Multispectral Instrument (MSI) is the sole scientific payload of the Sentinel-2 satellite. This is a high-resolution camera covering 13 bandwidths, spanning across the visible spectrum and also covering three bands in the near-infrared and shortwave infrared. The MSI is decomposed in several functional blocks that include the video compression unit. The video compression unit (VCU) contains the frontend electronics (FEE) system through with the MSI payload is controlled - this is a remote terminal (RT) in the Payload MIL-STD-1553B data bus. Besides the VCU, the MSI control also depends from the power conditioning and distribution unit (PCDU) and from the remote interface unit (RIU). In the case study described in Figure 1, the PCDU MSI (i.e. the VCU frontend) and the RIU are implemented by simple simulation models connected to the I/O Stubs & Adaptors. The MSI Control Application is the system under test and is the actual software deployed in the Sentinel-2 satellite.

Given this context, this section describes some of the challenges and high-level requirements that are addressed in this dissertation.

Simple Ontology of Testing Approaches

There are multiple approaches for testing: data oriented, keyword oriented, etc. In the sense intended here, a testing approach is a paradigm applicable to testing, in the same sense that: object oriented, service oriented and data oriented are paradigms applicable to software architecture. A paradigm has a set of benefits and limitations, and one or more application scenarios to which it is best suited for. An analysis will be provided showing the existing testing paradigms, including the features, limitations and reference application cases for each.

Robot Framework (Keyword-Driven Testing)

The Robot Framework is a technology implementing the keyword driven testing paradigm. In this dissertation we analyse the feasibility of using this framework for implementing the frontend test API. Should the feasibility be demonstrated by this initial analysis, shall validate it by implementing the existing test cases, which are an input to the proposed work, to actually validate the approach on a real-life case study.

Nature of the Aspects Tested Across Test Campaigns

When analysing the different test paradigms and the specific supporting technology, the different levels of testing and related aspects of the test environment shall be considered. For instance: module/unit testing, software integration, hardware/software (HW/SW) integration, system testing, acceptance testing, open-loop/closed-loop testing, etc.

The analysis the nature of the system under test shall also be considered. For instance, the SUT may be: a data-handling system or a control system, may have real time requirements, the order in which messages are exchanged may be important, the protocol messages may have a simple or complex structure.

The nature of the SUT and of the different test campaigns may determine, for instance, that a given test paradigm is effective and efficient for a given SUT type and/or test level and less effective and efficient for other scenarios.

Specific Aspects Driving Test Procedure Complexity

Which are the key factors that drive testers to develop complex test procedures? Is this a matter of technology alone - test procedures are complex because the test framework encourages complexity - or are there aspects of the SUT or test level, that impose the need for complexity. As guideline, possible aspects driving complexity may include: hard real-time behaviour, complex protocol data units, asynchronous behaviour or the need to perform robustness testing.

1.3 *Structure*

Besides this introductory chapter, this dissertation is organized in four main chapters:

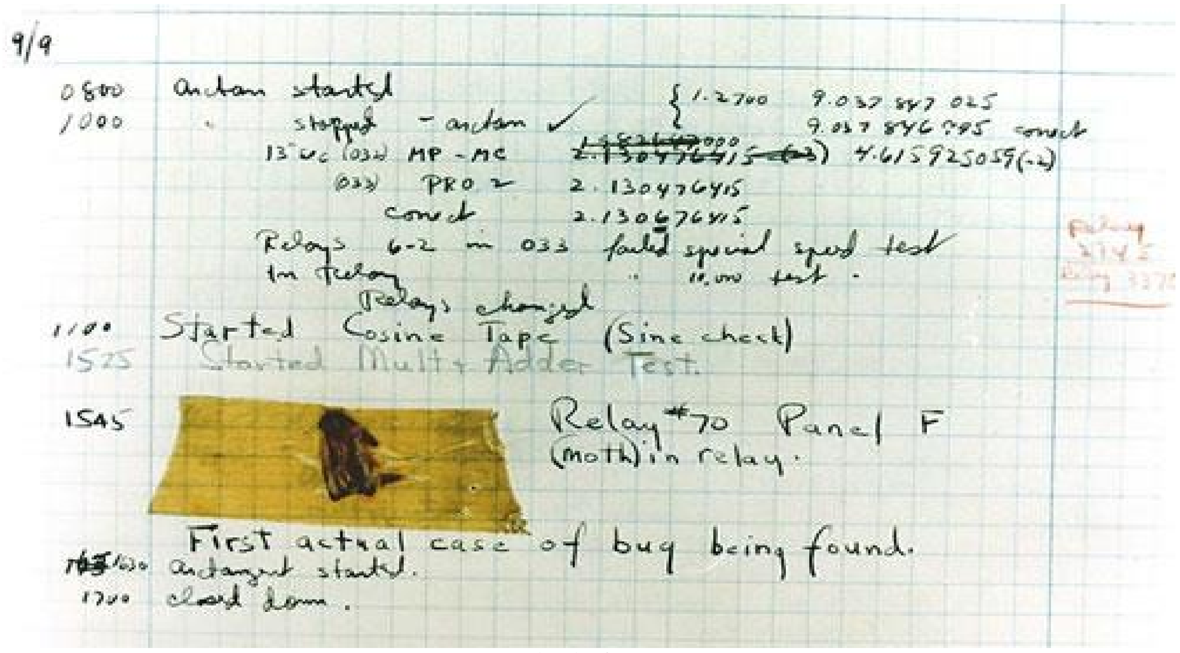
- Chapter 2 introduces the concept of bug. The definition of software bugs and the risks associated to them. After this, all the information about the domain of Software Testing is presented.
- Chapter 3 presents a detailed analysis of Satellites. Furthermore, the process of development and verification of these systems is presented.
- Chapter 4 starts by enumerating and explaining the objectives that were defined for the implementation of the API. Later, it describes the architecture that was conceptualized and implemented during this work, along with an explanation of how the components of the architecture interact with each other;
- Chapter 5 presents a satellite component, software requirements and a practical validation job in order to test a few features of this component using the API.
- Finally, Chapter 6, presents conclusions, cons and drawbacks provided by the API and future work.

2 State of the Art

2.1 *Bugs, the beginning*

In 1947, computers were big, room-sized machines operating on mechanical relays and glowing vacuum tubes. The state of the art at the time was the Mark II, a behemoth being built at Harvard University.

On 9th September, technicians were running the new computer through its paces when it suddenly stopped working. They scrambled to figure out why and discovered, stuck between a set of relay contacts deep in the bowels of the computer, a moth. It had apparently flown into the system, attracted by the light and heat, and was zapped by the high voltage when it landed on the relay. Grace Murray writes in her logbook with a moth taped in the page at 15:45: "Relay #70 Panel F (moth) in relay" and "First actual case of bug being found" [1] as depicted in Figure 2.



9/9

0800 Arctan started
1000 " stopped - arctan ✓ { 1.2700 9.037 847 025
9.037 846 995 correct
13" UC (033) MP - MC ~~1.982147000~~ ~~2.130476415 (-3)~~ 4.615925059(-2)
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 10,000 test .
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Arctangent started.
1700 closed down.

Relay
2145
Relay 3370

Figure 2 - Logbook of Grace Murray [2]

According with [1] a software bug occurs when one or more of the following five rules is true:

1) The software doesn't do something that the product specification says it should do.
2) The software does something that the product specification says it shouldn't do.
3) The software does something that the product specification doesn't mention.
4) The software doesn't do something that the product specification doesn't mention but should.
5) The software is difficult to understand, hard to use, slow, or - in the software tester's eyes - will be viewed by the end user as just plain not right.

It’s easy to take software for granted and not really appreciate how much it has infiltrated our daily lives. Back in 1947, the Mark II computer required legions of programmers to constantly maintain it.

Software is a key ingredient in many of the devices and systems that pervade our society. Software defines the behaviour of network routers, financial networks, telephone switching networks, the Web, and other infrastructure of modern life. Software is an essential component of embedded applications that control exotic applications such as airplanes, spaceships, and air traffic control systems, as well as mundane appliances such as watches, cars, phones, and remote controllers. Modern households have over 50 processors, and some new cars have over 100; all of them running software that optimistic consumers assume will never fail! [3] Software is everywhere. However, it’s written by people - so it’s not perfect [1].

To clarify the idea of danger associated to software bugs, some historical bugs are presented in attachment [Attachment A]. A bug can be an inconvenience when a game doesn’t work properly or it can even be catastrophic when it results in the loss of one or more human lives.

The search, the study, the observation of the results, the way a software behaves in the most diverse situations, even in unexpected situations, is called Software Testing, a branch of Software Engineering.

2.2 *Software Testing*

We can define Software Testing as a set of several processes: search for bugs, solve them, reduce the maintenance of a software as well reduce of cost of its development, still increase the guarantee of operation and quality for the client that receives the software.

According with Priya and Shukla [4] Software Testing means to cut errors, reduce maintenances and to short the cost of software development.

This process is not bounded to detection of “error” in software but also enhances the surety of proper functioning and help to find out the functional and non-functional particularities. When we speak about proper functioning, we refer to verify that a program gives a correct and an expected output based on specified input.

Another important topic presented by the authors [4], is the term quality. Testing is the process done to enhance the quality of software. Arriving to this topic we can observe that Software Testing is important to determine the quality of a software and is the major step in software engineering.

In terms of disadvantages, we can underline the high quantity of efforts and costs of this process. The authors advance that the process of testing takes between 40%-50% development efforts. This last sentence is even seen as a con by Glenford Myers [5]. Myers [5] states that software testing is the core component of Software Quality Assurance (SQA) and that numerous organizations and companies spend about 40% of their resources in this thematic. To justify this high percentage of efforts, Irena [6], uses "life-critical software" to demonstrate how critical and immune to software failures can/should be. Accompanying this high level of criticality and immunity is therefore accompanied by the high percentage of resources and costs to test software of this nature.

The main goal of software testing is systematically and stepwise detection of different classes of errors within a minimum amount of time and with a much less amount of effort.

Finally, we can observe the study produced by the ISTQB [7] with the aim of presenting standard information. At first glance, the most common realization about software testing is that it only consists of "running" tests, i.e. running the software. According to the document, the test activities are much more comprehensive and differentiated than the one presented previously. The test activities exist before and after the test execution phase. These activities include planning and control, choosing test conditions, designing and executing test cases, verifying results, and information on the test process and system being tested. The tests also include review documents (including source code).

Testing software from different points of view leads to different goals being considered. For example, when we are in the presence of component, integration or system tests, the ultimate goal is to create as many failures as possible so that software defects and errors are identified and then treated. In acceptance tests, the system is tested with the intention of checking if the system works as expected, to make sure that the system is in accordance with your requirements. On the other hand, we also have tests that its purpose not to search for and solve errors, but to evaluate the quality of the software. This process is important for evaluating and informing stakeholders (developer - client) about the risk of releasing a product at any given time.

2.2.1 Software Testing: Why?

According with ISTQB [7] there are five main subjects that explain why software testing is necessary and why is so important.

Software Systems Context

Software systems are an integral part of life, from business applications (e.g., banking) to consumer products (e.g., cars, smartphones). Most people have had an experience with software that did not work as expected. Software that does not work correctly can lead to many problems, including loss of money (Intel bug - Attachment A), time or business reputation, and could even cause injury or death (THERAC-25 - Attachment A).

How much testing is enough?

Testing should provide sufficient information to stakeholders to make informed decisions about the release of the software or system being tested, for the next development step or handover to customers.

Causes of Software Defects

A human being can make an error (mistake), which produces a defect (fault, bug) in the program code, or in a document. If a defect in code is executed, the system may fail to do what it should do (or do something it shouldn't), causing a failure. Defects in software, systems or documents may result in failures, but not all defects do so.

Defects occur because human beings are fallible and because there is time pressure, complex code, complexity of infrastructure, changing technologies, and/or many system interactions.

Role of Testing in Software Development, Maintenance and Operations

Rigorous testing of system and documentation can help to reduce the risk of problems occurring during operation and contribute to the quality of the software system, if the defects found are corrected before the system is released for operational use.

Software testing may also be required to meet contractual or legal requirements, or industry-specific standards.

Testing and Quality

With the help of testing, it is possible to measure the quality of software in terms of defects found, for both functional and non-functional software requirements and characteristic (e.g., reliability, usability, efficiency, maintainability and portability).

Testing can give confidence in the quality of the software if it finds few or no defects. A properly designed test that passes reduces the overall level of risk in a system. When testing does find defects, the quality of the software system increases when those defects are fixed.

Lessons should be learned from previous projects. By understanding the root causes of defects found in other projects, processes can be improved, which in turn should prevent those defects from reoccurring and, as a consequence, improve the quality of future systems. This is an aspect of quality assurance.

Testing should be integrated as one of the quality assurance activities (i.e., alongside development standards, training and defect analysis).

2.2.2 Objectives of Testing

In terms of objectives of this thematic we can highlight the four main objectives described by Myers [5] and the ISTQB documentation [7]:

Detection: Various errors, defects, and deficiencies are detected. System capabilities and various limitations, quality of all components, the work products, and the overall system are calculated.

Prevention: Prevent or reduce the number of errors. Different ways to avoid risks and to tackle problems in the future are identified.

Demonstration: It shows how the system can be used with various acceptable risks. It also demonstrates functions with special conditions and shows how products are ready for integration or use.

Improving quality: By doing effective testing on software, errors can be minimized and thus quality of software is improved.

Continuing in this topic, in [8] the authors refer that the basic purpose of software testing is Verification, Validation and Error Detection in order to find various errors and problems - and the aim of finding those problems is to get them fixed. The authors continue referring that Software Testing is more than just error detection. Software Testing is done under controlled conditions:

Verification: To verify if system behaves as specified. It is the checking and testing of items, which includes software, for conformance and consistency of software by evaluating the results against pre-defined requirements. In verification the question arises: "Are we building the product right?"
Validation: In this we check the system correctness which is the process of checking that what has been specified by user and what the user actually wanted. In validation the question arises: "Are we building the right system?"

Error Detection: to detect errors. A number of tests should be done to make things go wrong to determine if what things should happen when they should not.

As we can see above, Chaugan and Singh [8] present two important terms that Offutt [3] considers two of the most important terms in Software Testing. Offutt [3] describes validation being the process of evaluating software at the end of software development to ensure compliance with intended usage. Usually, depends on domain knowledge; that is, knowledge of the application for which the software is written. For example, validation of software for an airplane requires knowledge from aerospace engineers and pilots. On the other hand, verification is the process of determining whether the products of a given phase of the software development process fulfil the requirements established during the previous phase. Usually, is a more technical activity that uses knowledge about the individual software artefacts, requirements, and specifications.

2.2.3 Roadmap

As we already saw, software testing is more than run the software. The maximum that we can say is that the most visible part of testing is test execution. According with ISTQB [7], test plans should include time to be spend on planning the tests, designing tests cases, preparing for execution and evaluating results.

The fundamental test process consists of the following main activities illustrated in the Figure 3.

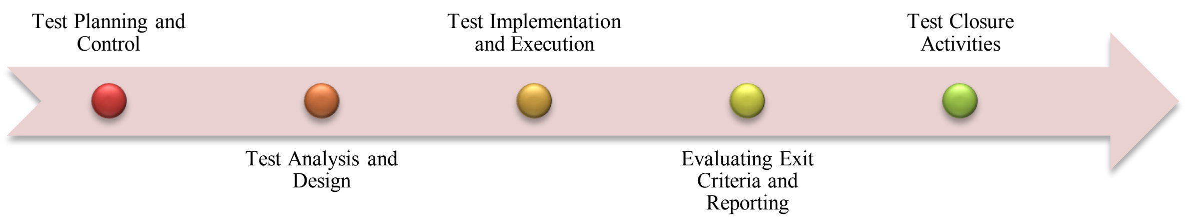

Test Planning and Control
Test Analysis and Design
Test Implementation and Execution
Evaluating Exit Criteria and Reporting
Test Closure Activities

Figure 3 - Fundamental Test Process

Although logically sequential, the activities in the process may overlap or take place concurrently. Tailoring these main activities within the context of the system and the project is usually required.

Test Planning and Control

Test planning is the activity of defining the objectives of testing and the specification of test activities in order to meet the objectives and mission.

Test control is the ongoing activity of comparing actual progress against the plan, and reporting the status, including deviations from the plan. It involves taking actions necessary to meet the mission and objectives of the project. In order to control testing, the testing activities should be monitored throughout the project. Test planning takes into account the feedback from monitoring and control activities.

Test Analysis and Design

Test analysis and design is the activity during which general testing objectives are transformed into tangible test conditions and test cases.

The test analysis and design activity comprehends the review of the test basis (such as requirements, software analysis reports, architecture, design, interface specifications), the evaluation of testability of test basis and test objects and the identification and prioritization of test conditions based on analysis of test items such as the specification, behaviour and structure of the software. Continuing the activity, the necessary data to support the test conditions and test cases is identified. Finally, the infrastructure and tools needed are identified and setup.

Test Implementation and Execution

Test implementation and execution is the activity where test procedures or scripts are specified by combining the test cases in a particular order and including any other information needed for test execution, the environment is set up and the tests run.

Evaluating Exit Criteria and Reporting

Evaluating exit criteria is the activity where test execution is assessed against the defined objectives. This should be done for each test level.

In this activity, the test logs are checked against the exit criteria specified in test planning, an analysis is conducted to assess if more tests are needed or if the exit criteria specified should be changed and a test summary report for stakeholders is written.

Test Closure Activities

Test closure activities collect data from completed test activities to consolidate experience, test ware, facts and numbers. Test closure activities occur at project milestones such as when a software system is released, a test project is completed (or cancelled), a milestone has been achieved, or a maintenance release has been completed. Test closure activities include tasks like, check which planned deliverables have been delivered, close incident reports or raising change records for any that remain open, document the acceptance of the system and finalize and archive test ware, the test environment and the test infrastructure for later reuse;

In order to gain most of the testing activities, a defined process must be followed. But before any testing activity begins, much of the effort should be spent on producing a good test plan. A good test plan goes a long way in ensuring that the testing activities are adhered to what the testing is trying to achieve.

Comparing these five activities, it is easy to see that the first two activities (Test Planning and Test Design) are intellectually challenging. Planning how much testing to do, determining appropriate completion criteria requires careful analysis and thought.

Similarly, specifying test cases (identifying the most important test conditions and designing good test cases) requires a good understanding of all the issues involved and skill in balancing them. These intellectual tasks govern the quality of test cases.

The next two activities (Test Execution and Test Reporting) involve predominantly administrative tasks. Furthermore, executing and recording are activities that are repeated many times whereas the first two activities, Test Planning and Test Design are performed only once (they may be revisited if the completion criteria are not met the first time around but they are not repeated from scratch).

2.2.4 Principals/Axioms

Several authors present a diversity of principles. They're axioms, truisms, or even aphorisms about the nature of software testing itself, although some are indeed paradoxical in nature. It is interesting to take a closer look at these realities, because each one of them offers a bit of knowledge that can help to put some aspects of the overall software testing process into perspective.

These principles are seen by the community as general ideas that professionals in this area point out from their experience. Some of them present more than others, depending on the degree of specificity each one use.

Making a cross reference between the ISTQB documentation [7], Ron Patton's book [1] and Glenford Myers [5] we can find several principles that have been reunited during the last 40 years and, as Ron Patton's [1] refers, should be understood as the "rules of the road" or "facts of life" for software testing:

It's impossible to test a program completely
In ISTQB [7], the document asserts that, the number of possible inputs, outputs, paths through the software is very large and the software specification is subjective. Considering this hugeness of possibilities, we obtain an even greater number of test conditions. A good example to describe this axiom is the Microsoft Windows Calculator.

If we want to test the Windows Calculator. Starting with addition, we try 1+0=. We get an answer of 1. That's correct. Then try 1+1=. Get 2. How far do we go? The calculator accepts a 32-digit number, so you must try all the possibilities up to 1+99999999999999999999999999999999=. Once we complete that series, we can move on to 2+0=, 2+1=, 2+2=, and so on. Eventually we'll get to 99999999999999999999999999999999+99999999999999999999999999999999=.

Next, we should try all the decimal values: 1.0+0.1, 1.0+0.2, and so on. Once that we verify that regular numbers sum properly, we need to attempt illegal inputs to assure that they're properly handled. Remember, we're not limited to clicking the numbers onscreen - we can press keys on computer keyboard, too. Good values to try might be 1+a, z+1, 1a1+2b2. There are literally billions upon billions of these.

Edited inputs must also be tested. The Windows Calculator allows the Backspace and Delete keys, so we should try them. 1<backspace>2+2 should equal 4. Everything we've tested so far must be retested by pressing the Backspace key for each entry, for each two entries, and so on. If we to complete all these cases, we can then move on to adding three numbers, then four numbers, …

There are so many possible entries that we could never complete them, even if we used a super computer to feed in the numbers. And that's only for addition. We still have subtraction, multiplication, division, square root, percentage, and inverse to cover.

The point of this example is to demonstrate that it's impossible to completely test a program, even software as simple as a calculator. If we decide to eliminate any of the test conditions because we feel they're redundant or unnecessary, or just to save time, we've decided not to test the program completely. This last sentence brings us to the following axiom.

Software Testing is a Risk-Based Exercise

Testing software with 100% sure is impossible [1]. If we decide not to test every possible test scenario, we've chosen to take on the risk. In the example of the calculator let's imagine that we choose not to test 1024 + 1024 = 2048? It is possible that the programmer accidentally left a bug in this case and the consequences can be very serious.

This may sound scary. We can't test everything, and if we don't, we will likely miss bugs. The product must be released, so we will need to stop testing, but if we stop too soon, there will still be areas untested. What do we do? When we should stop testing? What are the risks?

One key concept that software testers need to learn is how to reduce the huge domain of possible tests into a manageable set, and how to make wise risk-based decisions on what's important to test and what's not.

Ron Patton [1] presents a relationship between quantity of tests produced and number of bugs discovered. This relationship is illustrated in Figure 4.

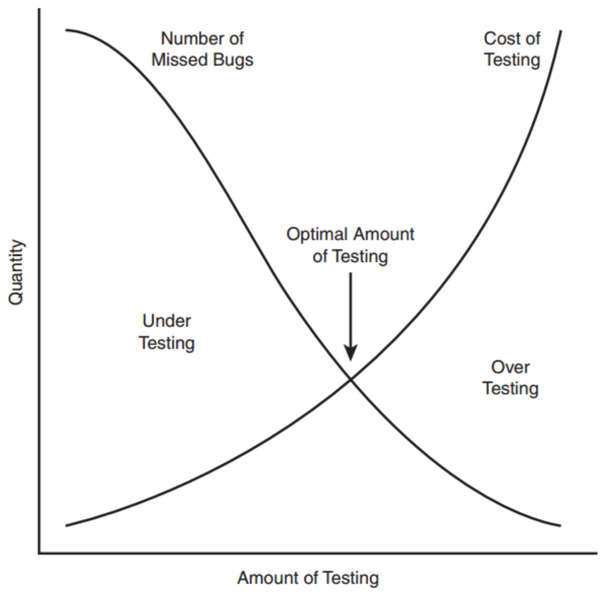


Figure 4 - Quantity vs Amount of Testing [1]

As depicted in Figure 4, we can see if we want to test everything, the cost will fire and the number of bugs to discover will drop to a minimum. Arriving at this minimum value, is not worth continuing to test as the cost vs effect is almost insignificant. On the other hand, if we adopt a strategy with a few tests or we take a line where we make bad decisions about what is important to test, the cost is minimal but the software will have a huge amount of bugs on it.

The best-case scenario, the intersection of the two lines, known as "Optimal Amount of Testing", correspond to the point where the number of tests is moderate so both the cost and the number of errors are not worrisome. We must balance between a responsible attitude in terms of cost and risks in terms of the number of bugs.

Testing Can’t Show That Bugs Don’t Exist

To better understand this principal, we came back to the term bug. In this case the real bug, the inset. Making the analogy, one exterminator charged with examining a house for bugs. He inspects the house and find evidence of bugs - maybe live bugs, dead bugs, or nests. He can safely say that the house has bugs.

He visits another house. This time he doesn’t find evidence of bugs. He looks in all the obvious places and see no signs of an infestation. Maybe he finds a few dead bugs or old nests but nothing that tells that live bugs exist. Can he absolutely, positively state that the house is bug free? Nope!! All he can conclude is that in his search he didn’t find any bugs alive. Unless he completely dismantled the house down to the foundation, he can’t be sure that he didn’t simply just miss them.

Software testing works exactly as the exterminator does. It can show that bugs exist, but it can’t show that bugs don’t exist.

We can also point out from Ron [1] analysis one “advice”: “We can perform tests, find and report bugs, but at no point can we guarantee that there are no longer any bugs to find. We can only continue our testing and possibly find more.”

The Pesticide Paradox

This phenomenon was presented by Boris Beizer in his book “Software Testing Techniques” [9] in 1990. The author asserted that software undergoing the same repetitive tests eventually builds up resistance to them, similar to the reaction of insects to pesticides: if he keeps applying the same pesticide, the insects eventually build up resistance and the pesticide no longer works.

When we have a test that removes one or more errors, running that same test over and over again will not eliminate errors that were previously removed, so the test becomes ineffective. Related to this, errors that remain get harder to detect. After several iterations, all the bugs that those tests would find have been exposed. Continuing to run them won’t reveal anything new.

To overcome the pesticide paradox, software testers must be willing to continually design new test cases that cover all or most of the scenarios where bugs might be present. They might also adopt a methodology that allows them to reuse the test code to automatically test new scenarios and code paths (better known as model-based testing) [1] and [7].

Not all the bugs found will be fixed

One of the sad realities of software testing is that even after all hard work, not every bug will be fixed. This doesn’t mean that the tester failed in achieving the goal as a software tester, nor does it mean that he or the team will release a poor-quality product. It means, however, that he will need to rely on an exercising good judgment and knowing when perfection isn’t reasonably attainable. Risk-based decisions for each and every bug will be needed, deciding which ones will be fixed and which ones won’t.

The decision-making process usually involves the software testers, the project managers, and the programmers. Each carries a unique perspective on the bugs and has his own information and opinions as to why they should or shouldn't be fixed [1].

The purpose of paradoxes is to capture attention and to provoke fresh thought. In science, this process frequently leads to major breakthroughs. While we don't claim that the testing paradoxes described above will lead to major breakthroughs within testing, we would argue that becoming aware of them leads us to think about the matter - and thinking may lead to creative rethinking. Maybe we can expand our focus to look at options or possibilities that we normally wouldn't consider. Maybe that process will turn a potentially negative outcome into something positive. Or - maybe we should just view these phenomena as little particles of knowledge that can help us put some aspects of the overall software testing process into clearer perspective.

2.3 *Levels of Testing*

To enhance the quality of software testing, and to produce a more unified testing methodology applicable across several projects, the testing process could be abstracted to different levels according the development methodology. The Figure 5 shows the arrangement and relationship of the different levels.

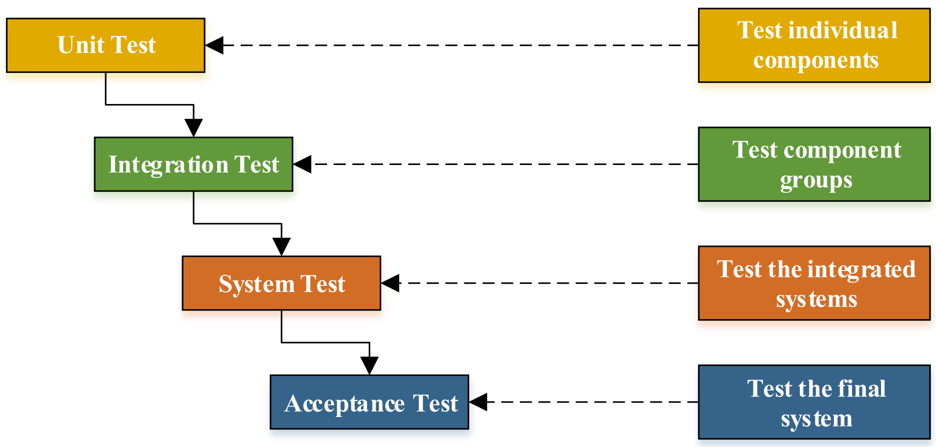


Figure 5 - Software Testing Levels

The levels of testing (except for regression testing which is considered a re-testing) have a hierarchical structure which builds up from the up down - where lower levels assume successful and satisfactory completion of higher level tests.

It is normally required for unit testing to be done before integration testing which is done before system testing and acceptance testing. Each level of test is characterised by an environment i.e. a type of people, hardware, software, data and interface. For example, the people involved in unit testing are mainly programmers, while testers are mainly those involved in system testing.

2.3.1 Unit testing

To understand the concept of this level we need to know what a unit is. A unit is the smallest piece of software going for testing [9], i.e., smallest set of lines of code which can be tested [8]. In procedural programming, a unit may be an individual program, function or procedure. In object-oriented programming, the smallest unit is a method, which may belong to a base/super class, abstract class or derived/child class.

Knowing this definition, we can easily imply that this level of testing focus in the smallest testable part of an entire application. The main purpose of this testing is to ensure that a particular unit or module is working according to functional specifications [10] and [11].

This level, also known as component, module or program testing, searches for defects in, and verifies the functioning of, software modules, programs, objects and classes, that are separately testable. It may be done in isolation from the rest of the system, depending on the context of the development life cycle and the system. Stubs, drivers and simulators may be used.

Component testing may include testing of functionality and specific non-functional characteristics, such as resource-behaviour (searching for memory leaks) or robustness testing, as well as structural testing (decision coverage).

Typically, component testing occurs with access to the code being tested and with the support of a development environment, such as a unit test framework or debugging tool. In practice, component testing usually involves the programmer who wrote the code. Defects are typically fixed as soon as they are found, without formally managing these defects [7].

Unit testing is just one of the levels of testing which contribute to make the big picture of testing a whole system [8]. When we start a unit test in parallel with development it may look like a slow process as many defects are uncovered during this stage and several changes are made to the code. However, with time the code is refined and the number of defects begins to reduce. So, the foundation of the software is strong and in the later stages the software development is carried out at a much faster pace thereby saving a lot of time.

If the unit testing is carried out properly then it would also result in a lot of cost saving as the cost of fixing a defect in the final stages of software development are much higher than fixing them in the initial stages [12]. To endorse this last sentence, we can take a look on the article publish by Rudra Infotech in [13]. In the article, we can observe how the cost of fixing a bug evolves according the phase of the Software Development Life-Cycle (SDLC).

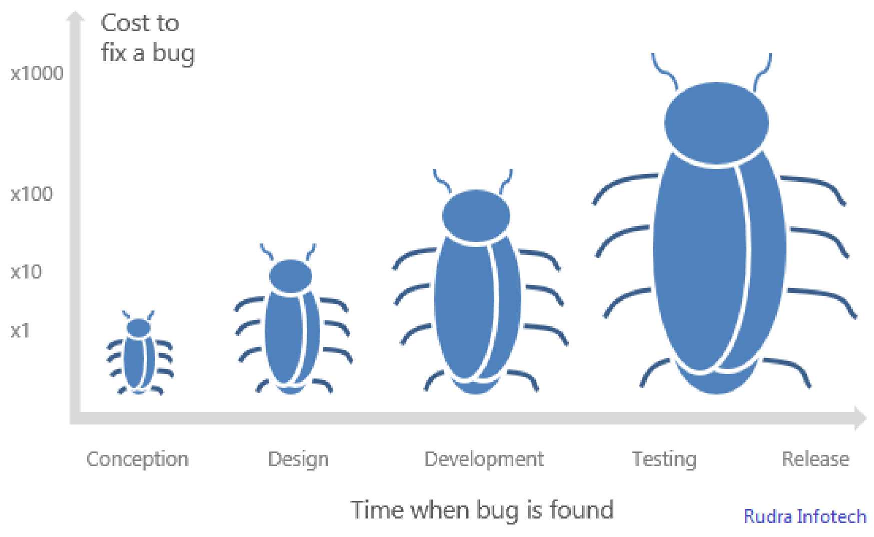


Figure 6 - Times when bug is found vs Cost to fix a bug

Analysing Figure 6 presented by Rudra Infotech in [13], a bug identified during conception costs something around zero, but when this same bug is found only after implementation or test, the average cost of repair can get to something between 10 and 1000 times more than in the previous step. When customers find this bug in production environment, the cost of the problem considers all side effects related to it and. That is where things can get serious.

Summarizing, we can use these thoughts and apply them in the software testing levels. The cost of fixing bugs on unit level certainly will be less than integration level, the cost of fixing bugs on integration will be less than system level and so on.

2.3.2 Integration testing

As the name suggests, this level of test involves building a system from its components and testing it for problems that arise from component interactions. To simplify error localization, systems should be incrementally integrated.

In integration testing, the code is divided into individual segments and tested as a group. The main task of integration testing is to technically verify proper interfacing between modules, and within sub-systems. In other words, detect faults amongst the interaction between integrated units.

Integration level follow up the unit test. Is a technique that systematically construct the program structure while at the same time conduct tests to uncover errors associated with interface. The idea is to take unit tested components and build a program structure that has been dictated by design [14].

According with ISTQB [7] we have more than one level of integration testing and it may be carried out on test objects of varying size as follows:

Component integration testing: tests the interactions between software components and is done after component testing.

System integration testing: tests the interactions between different systems or between hardware and software and may be done after system testing.

Through the last sentence we can observe a constraint of this level. The grater the scope of integration, the more difficult it becomes to isolate defects to a specific component or system, which may lead to increased risk and additional time for troubleshooting.

2.3.3 System testing

Once the integration testing phase gets successfully completed it is time to move on to system testing where the system as a whole, with all components well integrated, is ready for further testing. This testing is used for requirement analysis, verifies that weather the system is working according to requirement specification or not [10]. It helps in discovering the confidence that a quality product is delivered. Apart of requirement analysis, non-functional quality attributes, such as security, reliability, and maintainability, are also checked [8]. For this form of testing it is very important to create a scenario similar to the real-time scenario where the system will be deployed.

System testing may include tests based on risks and/or on requirements specifications, business processes, use cases, or other high level test descriptions or models of system behaviour, interactions with the operating system, and system resources [7].

The system testing has some objectives such as, verify that the system components perform control functions, perform inter-system test, demonstrate that the system performs both functionally and operationally tasks as specified and perform appropriate types of tests related to transaction flow, installation and reliability [15].

2.3.4 Acceptance testing

This testing is done when the complete system is handed over to the customers or users from developer side.

Acceptance testing is also known as validation testing, final testing, QA testing, factory acceptance testing and application testing. And in software engineering, acceptance testing may be carried out at two different levels; one at the system provider level and another at the end user level [14]. The aim of acceptance testing is not to find out simple errors, cosmetic errors and spelling mistake but also to find bugs in whole system that will lead to system failure and application crash. The acceptance testing is used to ensure that the application is acceptable for delivery or not [4] and [8].

2.3.5 Regression testing

When any modification or changes are done to the application or even when any small change is done to the code then it can bring unexpected issues. Along with the new changes it becomes very important to test whether the existing functionality is intact or not. This can be achieved by doing the regression testing.

The purpose of the regression testing is to find the bugs which may get introduced accidentally because of the new changes or modification. When some defect gets fixed there might be a possibility that the fix may have introduced or uncovered a different defect elsewhere in the software. The way to detect these 'unexpected side-effects' of fixes is to do regression testing. In practice, regression test is not considered a different level of testing. It is "re-testing" [4].

This testing becomes very important when there are continuous modifications or enhancements done in the application or product. These changes or enhancements should NOT introduce new issues in the existing tested code. This helps in maintaining the quality of the product along with the new changes in the application [4], [15] and [11].

The regression testing can be done by automation tools that can decrease the effort of testers otherwise, can be very tedious and time consuming since the same set of test cases are executed again and again.

2.4 *Software Testing Methods*

2.4.1 White-Box

White-box testing test the internal structure or working of an application. This method is highly effective in finding errors and bugs in the program. In white-box testing tester uses specific knowledge of the program to verify the output.

This method is also called glass box testing, clear box testing, open box testing, transparent box testing, structural testing, logic driven testing and design based testing [16].

According with R. Kaur [8], in this method, internal details and structure of the system are visible. Thus, it is highly efficient in detecting and resolving problems, because bugs can often be found before they cause trouble. Similarly in [16] and [4], the author assess that this method it's a strategy for finding errors in which the tester has complete knowledge of how the program components interact, i.e., the knowledge of internal structure and coding. Because of this knowledge K. Mohd [16] explains that this method can uncover implementation errors such as poor key management. The author ends saying that white box testing is applicable at unit, integration and system levels of the software testing process.

One conclusion that we can take from this method is when failures are discovered. The discovery of failures may lead to changes that require all white-boxes testing to be repeated [4]. Besides, in [17], white-box is considered as a security testing method that can be used to validate whether code implementation follows intended design, to validate implemented security functionality, and to uncover exploitable vulnerabilities.

Analysing several authors [16], [4], [6] and [18] we can point out several advantages and disadvantages of this method. Since the testers have complete knowledge of the code, there's a natural desire to exercise all paths, logical decisions, structures and loops. This goal will be impossible to achieve. First, it's very expensive as it requires a skilled tester to perform it, secondly, execute all possible logic paths would entail in an astronomically large number of tests and third, the complexity will be huge.

The adopted policy should test, at least one time, all the paths, loops or structures. With this policy, errors will be revealed and the tester can help in the optimization of code.

2.4.2 Black-Box

As the name suggest, in this method of testing the code is not accessible or even view, we only have a "black box". A black box is any device whose internal details and workings are not understood by or accessible to its user. It's testing of software based on specifications and output requirements and without any knowledge of the coding or internal structure in the program [8].

This method is also known as opaque testing, functional testing, close box testing and behavioural testing [19].

This type of testing is based on the output requirement of the system with any knowledge of coding in program or internal structure. In other words, this type of testing is done to examine the functionality of an application without concern about the internal working of the software. When a tester uses this method, will work on system's user interface by given inputs and check for the outputs without know about the coding and other inner working of code. It checks for all possible combination of end users [4].

Similarly, to P. Shukla and D. Mishra, Lewis in [18] explains this thematic with a good example. The author explains that this method of testing is based on the program or systems functionality. That is, the tester requires information about the input data and observed output, but doesn't know how the program or system works. Just as someone doesn't have to know how a car works internally to drive it, it's not necessary to know the internal structure of a program to execute it. The tester focuses on testing the program's functionality against the specification. It's completely unconcerned with the internal structure of the program or system.

The main aim is to test how well the system conforms to the specified requirements for the system. Black-box testing have little or no knowledge to the internal logical structure of the system. Thus, it only examines the fundamental aspect of the system. It makes sure that all inputs are properly accepted and outputs are correctly produced [20] and [8].

2.4.3 Grey-Box

In recent years, a third testing method has been also considered, the grey-box method. It uses internal data structures and algorithms for designing the test cases more than black-box testing but much less than white-box testing [8].

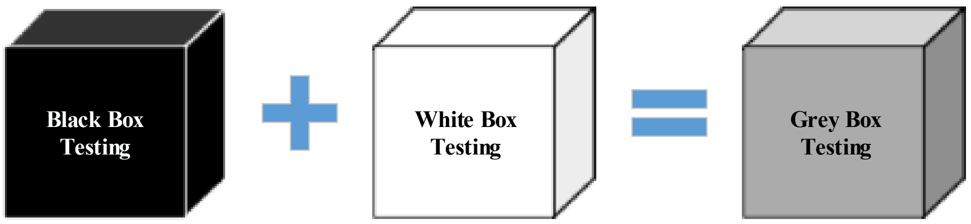


Figure 7 - Grey-Box Testing

As depicted in Figure 7, Grey-box method combines the testing methodology of White-box and Black-box. Grey-box testing technique is used for testing a piece of software against its specifications but using some knowledge of its internal working as well [21]. R. Kaur [8] considers this method unbiased and non-intrusive because it doesn't require that the tester have access to internal source code.

In attachment, a table with the principal points of each method can be found [Attachment B].

2.5 *Software Testing Strategies*

2.5.1 Manual Strategy

In this strategy, the tester produces and run test cases manually [15], trying a variety of input combinations, comparing the program expectations and the actual outcomes in order to find software defects. These manual tests are no more than the tester using the program as an end user would, and then determining whether or not the program acts appropriately [22].

This strategy can be viewed as a process, as depicted in the Figure 8. The test team starts to generate various test cases, take the software image, and execute each test case to test all functionalities. If a defect is found, a bug report is prepared, sent to the project manager, test manager and to the programmer. The software is modified and the same steps are repeated again till the error is removed [23].

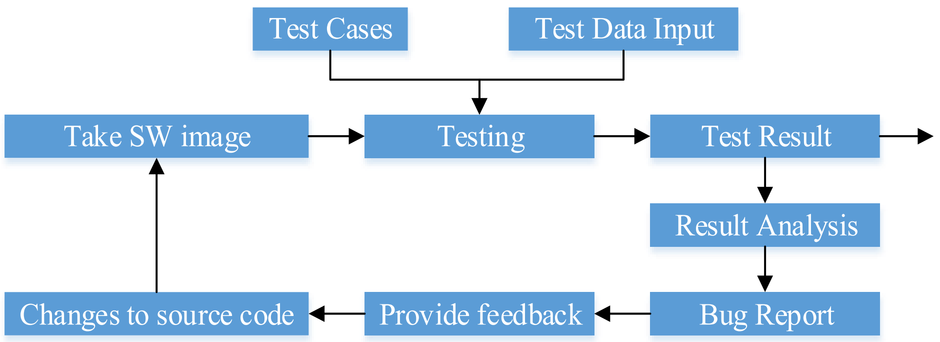


Figure 8 - Manual Testing [25]

2.5.2 Automated Strategy

Organisations often seek to reduce the cost of testing. Most organizations aren't comfortable with reducing the amount of testing so, instead, they look at improving the efficiency of testing. Luckily, there are a number of solutions that are claimed to be able to do just this. There are automated tools which take a test case, automate it and run it against a software target repeatedly [24].

The automated philosophy is simple, automate the manual testing process currently in use. Automation is the use of scripts and tools that reduce the need of manual or human involvement or interaction in repetitive or redundant tasks [25] and [26]. With the last sentence, we can have the feeling that the objective of automation is to eliminate testers (persons). Instead, it should aim to help them to make better use of their time. If automation is not well supported by the manual testers themselves, it might cause a kind of deprecation in the quality of the product.

Automating software involves developing test scripts using scripting languages such as Python or JavaScript, so that test cases can be executed by computers with minimal human intervention and attention [22], [25] and [14]. These scripts and tools can also feed test data to the system under test, compare expected and actual results and generate detailed test reports [25].

Test automation can be used in multiple ways. It can and should be used differently in different contexts and no single automation approach works everywhere. Test automation is no silver bullet either but, it has a lot of potential and when done well it can significantly help test engineers to get their work done [27].

2.5.3 Automation: Why?

Companies not only want to test software adequately, but also as quickly and thoroughly as possible. To accomplish this goal, organizations are focused more and more in automated testing. Apart of all advantages, we will discuss the key concepts, presented in [27], [28], [29], [30], [31] and [32], that justify why automation it's becoming more and more used compared to the manual approach.

The first key concept is scalability. Taking the example of websites like Amazon or Google, large number of users are expected to visit these websites simultaneously. The biggest challenge for the software team is to ensure consistency and continuation of services even during peak loads. This means, that the website must be tested with large number of users (thousands or even millions!) to simulate the real-life scenario. It's not feasible for any software organisation to simulate these scenarios with the actual users. This is only possible with the help of automation tools. Tools can simulate any number of users.

Secondly, we have efficiency. As already presented, automated testing is faster than manual. Automated software testing can reduce the time to run repetitive tests from days to hours. Saving time translates directly into cost savings so, we will increase efficiency [33].

The biggest problem with the manual testing is the accuracy of the testing, making this concept one of the most important. Humans are prone to make mistakes: "Who never failed to shoot the first stone". This fact accompanied with repetitive test cases and monotonous tasks with long periods of time is shore that nothing good and positive will came. Tools are not prone to such errors, and hence, can produce accurate results always. These tools can run the same testing steps repeatedly without making any mistake [33] and [34].

The final key concept is related to test coverage. Coverage of testing refers to the number of test cases covering structural and functional testing, multiple environments and memory tests. With the use of testing tools, project teams can get better coverage. For example, test scripts created for one browser can run on multiple browsers without any changes. The testing tool provides this feature. Similarly, test scripts written for one environment can run on multiple environments [33].

2.5.4 Cost estimation between Manual and Automated strategies

In many industrial projects, the estimates conducted are limited to considerations of cost only. In many cases the investigated costs include costs for the testing tool or framework, labour costs associated with automating the tests and labour costs associated with maintaining the automated tests.

These costs can be divided into fixed and variable costs. Fixed costs are the upfront costs involved in test automation. Variable costs increase with the number of automated test executions.

In [35], a case study originally published by Linz and Daigl [36], details the costs for test automation as follows:

$V \equiv$ *cost of test specification and implementation*; $D \equiv$ *cost of a single test execution*

Accordingly, the costs for a single automated test (A_a) can be calculated as:

$$A_a = V_a + n * D_a$$

V_a - cost for specifying and automating the test case

D_a - cost for executing the test case one time

n - number of automated test executions

Following this model, to calculate the break-even point for test automation, the cost for manual test execution of a single test case (A_m) is calculated similarly as:

$$A_m = V_m + n * D_m$$

V_m - cost for specifying the test case

D_m - cost for executing the test case

n - number of manual test executions

The break-even point for test automation can then be calculated by comparing the cost for automated testing (A_a) to the cost of manual testing (A_m) as:

$$E(n) = \frac{A_a}{A_m} = \frac{(V_a + n * D_a)}{(V_m + n * D_m)}$$

According to this model, the benefit of test automation seems clear: "From an economic standpoint, it makes sense to automate a given test only when the cost of automation is less than the cost of manually executing the test the same number of times that the automated test script would be executed over its lifetime." Figure 9 depicts this interrelation. The x-axis shows the number of test runs, while the y-axis shows the cost incurred in testing. The two curves illustrate how the costs increase with every test run. While the curve for manual testing costs is steeply rising, automated test execution costs increase only moderately. Automated testing requires a much higher initial investment than manual test execution does ($V_a > V_m$).

According to this model, the break-even point for test automation is reached at the intersection point of the two curves. This "universal formula" for test automation costs has been frequently cited in software testing literature (e.g. [35], [37], [27]) and studies to argue in favour for test automation.

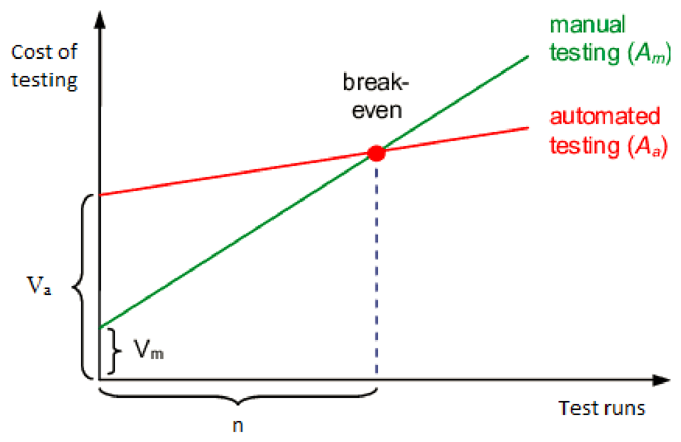


Figure 9 - Break-even point for Automated Testing

Depending on the author, the quoted number of test runs (n) required to reach the break-even point varies from 2 to 20. With this simple analysis, we can have a clear idea related to costs using Manual or Automated strategy. The logic of this formula is appealing and - in a narrow context - correct. "As a simple approximation of costs, these formulas are fair enough. They capture the common observation that automated testing typically has higher upfront costs while providing reduced execution costs" [38].

With this example, we can produce a forecast related to the cost of using a manual or automated strategy. Another example cited in literature is the study performed by Sabev and Grigorova [39].

2.5.5 Automation Testing Frameworks

A test automation framework is a set of assumptions, concepts, best practices and tools that provides support for automated software testing [28], [29] and [40]. Simply, is the system in which the tests will be automated.

It is important for every project team to define the goals of using an automation framework. This helps the project team in setting up the correct expectations, and accordingly, resources can be allocated. Some of the goals and objectives of using an automation framework can be identified such as write and run test cases with minimal or no scripting, efficiency of running repeatable test cases and ability to run test cases on multiple platforms with minimal effort.

A testing framework has six components, as depicted in Figure 10.

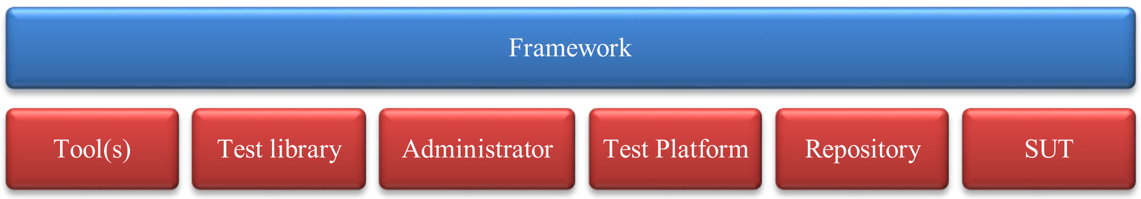


Figure 10 - Components of Automation Framework

The first component represents the testing tool(s). There's a high variety of tools with a variety of purposes available. Tools for writing and executing test scripts, tools for bug reporting and tools for storing test scripts are some of the examples. The project team evaluates the suitability and cost-benefit analysis and decides the usage of tools.

In second place, we can see the testing case library. All the necessary intelligence is built in the function library so that it can read the excel sheets and call the different functions from the function libraries itself based on the keywords for example.

Every automation framework needs an administrator to manage all the components and specifically test case libraries, test platforms, test tools, maintain the project templates, provide usage help and provide assistance on writing the test scripts.

The next component is the testing environment itself. This environment includes hardware, software and the testing tools. Also include the network connectivity for all the testing clients. Without a proper testing environment, it will be difficult to carry out proper testing.

During the testing process, the tester needs to store all the objects that will be used in the scripts in one or more centralized locations rather than letting them be scattered all over the test scripts. This is why a repository is needed. A repository is a file that maintains a logical representation of each application object that is referenced in an automated script.

Finally, the most important component is the system under test. The automation testing framework is always developed based on the system under test and his nature is the major factor to decide the testing approach.

The usage of automation frameworks can be advantageous since the long-term maintenance cost is low, it's easy to expand, maintain and perpetuate and the frameworks ensures that the testing effort is consistent because of standard libraries, compliance to standards, etc. However, this can be expensive since the framework need to be developed by someone and time is needed to train the team.

2.5.6 Types of Automation Frameworks

There are different types of models for test automation frameworks - for example, some of them are keyword oriented, where a table of keywords provides the basis for building test cases. A data-driven approach is also possible, where the test framework supplies "inputs" and observes a series of corresponding "outputs." The most common types of testing frameworks are:

Linear framework

In this framework, the scripts are written in a step-by-step manner as depicted in the test case flow. No functions are created and all the steps are written one after the other in a linear fashion. This framework is also known as “Record & Playback”. Tester manually records each step (navigation and user inputs), inserts checkpoints (validation steps) in the first round. He then plays back the recorded script in the subsequent rounds [41].

Modular framework

The modular framework can be resumed in one sentence: “Division of Linear Framework script in different parts” [33]. To implement the modularity, we basically divide the test in different parts so that we can form functions for reusing. These small scripts are then used in a hierarchical fashion to construct larger tests, realizing a particular test case. The test scripts of modularity framework use the principles of encapsulation, abstraction and inheritance, which are object-oriented programming’s (OOP’s) concepts. The application of these principles improve the maintainability and scalability of automated test suites [40]. Therefore a Modular framework is suitable for automation of large, stable applications [34].

Keyword-driven automation framework

The keyword-driven framework deals with functions [33]. A function is an important part of programming allowing the creation of chunks of code that performs a specific task. Keyword-driven testing is also known as table-driven testing or action-word testing. The basic idea of this framework is create functions forming the function library and call them as keywords in the tests, each keyword is associated with separated functions.

In [40], Michael Kelly affirm that keyword-driven tests look very similar to manual test cases. This is explained because the framework 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 and the data.

Another thing to keep in mind is that the initial investment is pretty high, the benefits of this can only be realized if the application is considerably big and the test scripts are to be maintained for quite a few years.

Data-driven automation framework

Data-driven testing automation framework is a type of framework which is dependent on data, which means that test data is the important factor here. The test input data and the expected output data are kept in separate files where a test driver script can execute all the test cases with input data and compare it with the expected output results. The test driver script also logs the status of execution of the test scripts by comparing the actual results with the expected output results. Figure 11 shows how a data-driven framework works:

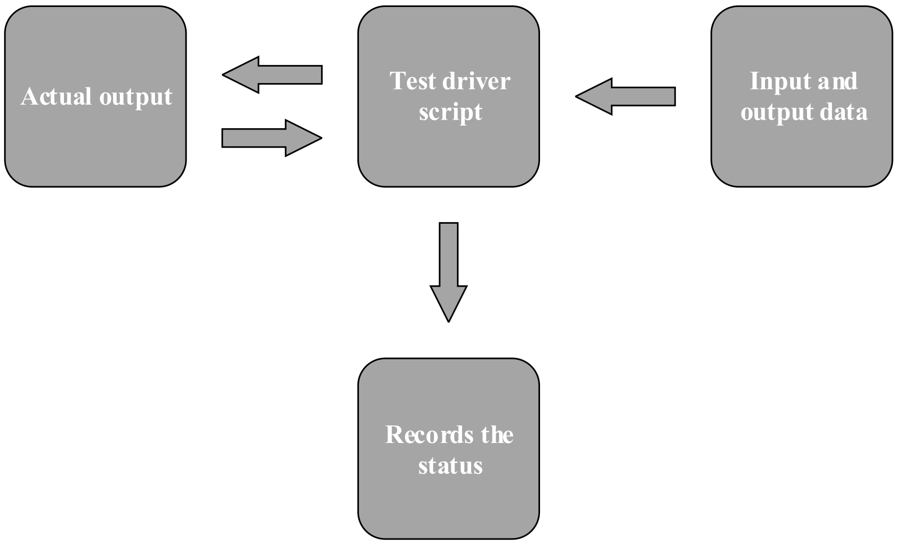


Figure 11 - Data-Driven Framework

The test scripts once written can be run any number of times by using the input and output dataset. The tests can be repeated using the pre-defined datasets without impacting the last cases [40].

This is similar to table-driven testing in that the test case is contained in the data file and not in the script; the script is just a “driver”, or delivery mechanism, for the data.

Hybrid automation Framework

Hybrid automation framework is a combination of modular, keyword and data-driven automation frameworks [34]. This is also the most commonly used approach for automation testing. The reason behind using this approach is to use the strengths of each framework and overcome the weaknesses of each one [40]. So, the hybrid automation framework has the following key points:

- Modularisation is used (from the modularity approach) to design the test cases and scripts. But we avoid combining of input data with the scripts as used in the modularity approach.
- Data-driven or keyword-driven approach is used to separate the input data from the test script. Depending on the availability of the resources and complexity of the project, data-driven or keyword-driven approach is used for testing.

2.6 *Test Automation Tools*

As we saw, automated testing brings us some benefits like short the development cycles, avoid cumbersome repetitive tasks and improve software quality. There are many test automation tools available in the market, divided in two types [42], [43], [44] and [45]. On one side, we have the open source test tools that, can be downloaded from the internet or can be obtained by the vendor without any charge. Users can use the tools at free of cost. Or, in other side, there's the commercial ones, the commercial software for sale. User should pay for it to use the software. Costs may be as per the functionality of the test tool.

In our analysis, our focus will be the biggest tools available in the market: IBM Rational Functional Tester, HP Quick Test Pro and Test Complete. Apart from these, we will introduce VectorCAST and Robot Framework too. The first four tools, will be presented regarding its dimension and long path in testing embedded systems. Regarding Robot Framework, this tool will be presented because of its increasing use and reputation and, as the master thesis proposal highlights, its capabilities to use keyword-driven approach. This tool was also chosen to implement the API that is intended to achieve in the end of this work.

IBM Rational Functional Tester

IBM Rational Functional Tester is an automated functional testing and regression testing tool. This software enables automated testing capabilities for regression, functional, GUI, and data-driven testing. Rational Functional Tester is an object-oriented automated testing tool that tests VB.NET, HTML, Java and Windows applications, and record robust and reliable scripts that can be played back to validate new builds of a test application. The recording mechanism creates a test script from the actions. Test scripts can then be executed by Rational Functional Tester to validate application functionality. Rational Function Tester supports a range of applications, such as web-based, .Net, Java, Siebel, Power Builder, Ajax, GEF, Adobe PDF documents.

HP Quick Test Pro (QTP) or HP Unified Functional Testing (UFT)

HP UFT is a functional testing tool from HP which is best suited for regression testing of the applications [44] and [46]. Quick Test Professional is a graphical interface record-playback automation tool [42] and [43].

Automated testing tool QTP provides good solutions for functional test and regression test automation. Quick Test Professional also enables test Java applets and applications, and multimedia objects on applications as well as standard Windows applications, Visual Basic applications and .NET framework applications. It works by identifying the objects in the application user interface or a web page and perform desired operations (such as mouse clicks or keyboard events).

HP Quick Test Professional uses a VBScript scripting language to specify the test procedure and to manipulate the objects and controls of the application under test. To perform more sophisticated actions, users may need to manipulate the underlying VBScript.

HP Quick Test also offers a fresh approach to automated testing: It deploys the concept of keyword-driven testing to radically simplify test creation and maintenance [47].

VectorCAST

VectorCAST embedded software testing platform is a family of products that automates testing activities across the software development lifecycle. VectorCAST also automates the tasks associated with unit, integration, and system testing of C, C++, and Ada applications, resulting in measurable reductions in cost and measurable improvements in quality [48].

Regarding VectorCAST products we can summarize VectorCAST/C++ that is a highly automated unit and integration test solution used by embedded developers to validate safety and business critical embedded systems. This dynamic test solution is widely used in the avionics, medical device, automotive, industrial controls, railway, and financial industries. Another product is VectorCAST/Ada, a dynamic software test solution that automates Ada unit and integration testing, which is necessary for validating safety - and mission-critical embedded systems.

Test Complete

Test Complete provides special features for automating test actions, creating tests, defining baseline data, running tests and logging test results. Like QTP, this tool is considered a graphical interface record-playback automation tool [43]. We just need to start recording, perform all the needed actions against the tested application and the tool will automatically convert all the "recorded" actions to a test.

The tool was developed by Smartbear software that gives testers the ability to create automated tests for Windows, Web, Android, and IOS applications. Test Complete supports various testing types and methodologies: unit testing, functional and GUI testing, regression testing, distributed testing.

This tool helps to keep the balance between quality and speed of delivery of applications at affordable cost [49].

Robot Framework

Robot Framework is a Python-based and an extensible keyword-driven test automation framework for end-to-end acceptance testing and acceptance-test-driven development (ATDD) [50]. The framework is free to use and published in compliance with Apache License 2.0.

It uses a keyword-driven testing (already presented). The keywords in the library are written either in Python or in Java. The text syntax follows a tabular style which makes writing test cases more user-friendly. The framework allows us to perform system testing, acceptance testing and regression testing. The software supports a high-level structure of tests and provides multiple test editors such as RIDE or Eclipse plugin so that users can easily maintain and scale the tests [50].

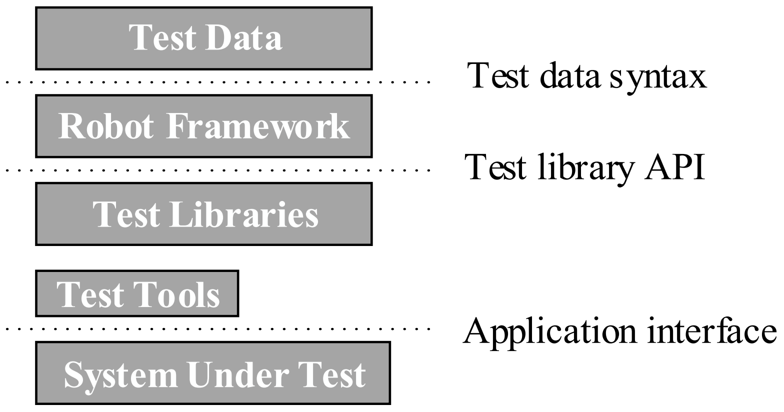


Figure 12 - Robot Framework Architecture [50]

The Figure 12, illustrates the high-level architecture of Robot Framework. It receives the test data and uses test libraries to communicate with the system that is being tested. The interaction between test libraries and the system under test is usually direct. However, some test libraries need to have drivers such as Selenium2Library in order to connect with the SUT.

The framework contains many standard and external libraries to support various kinds of testing. Each library serves a unique testing purpose. Standard test libraries are included while installing Robot Framework such as Builtin, Operating System, Dialogs and Remote. On the other hand, external libraries are created to meet the user's desires and requirements to perform certain testing purposes.

One great advantage of the tool is that new core keywords can be written to fill certain needs, which enhances the testing capabilities of the tool.

When we start to search for the right automated software testing tool, it's important to create a list of requirements about our needs. If we don't have a list of requirements, we may waste time downloading, installing and evaluating tools that only meet some of requirements or may not meet any of them. According to several authors, these requirements should address the test tool characteristics, test evaluating capability, scripts reusability, play back capability, and vendor qualification [43].

In attachment [Attachment B], a comparison study between the tools presented is available.

THIS PAGE INTENTIONALLY LEFT BLANK

3 Satellites, Development and Verification

3.1 *What Is a Satellite?*

A satellite is a moon, planet or machine that orbits a planet or star. For example, Earth is a satellite because it orbits the sun. Likewise, the moon is a satellite because it orbits Earth. Usually, the word "satellite" refers to a machine that is launched into space and moves around Earth or another body in space [51].

Thousands of artificial, or man-made, satellites orbit Earth with a range of functions. We can highlight several satellite applications such as: weather monitoring, navigation assistance, communications, Earth and space observation and national security.

The first artificial satellite was Sputnik, a Russian beach-ball-size space probe that lifted off on October 4, 1957. That act shocked much of the western world, as it was believed the Soviets did not have the capability to send satellites into space [51].

3.2 *The importance of satellites*

The bird's-eye view that satellites have allows them to see large areas of Earth at one time. This ability means satellites can collect more data, more quickly, than instruments on the ground [51].

Satellites also can see into space better than telescopes at Earth's surface. That's because satellites fly above the clouds, dust and molecules in the atmosphere that can block the view from ground level.

Before satellites, TV signals didn't go very far. TV signals only travel in straight lines. So, they would quickly trail off into space instead of following Earth's curve. Sometimes mountains or tall buildings would block them. Phone calls to faraway places were also a problem. Setting up telephone wires over long distances or underwater is difficult and costs a lot. With satellites, TV signals and phone calls are sent upward to a satellite (uplink). Then, almost instantly, the satellite can send them back down (downlink) to different locations on Earth.

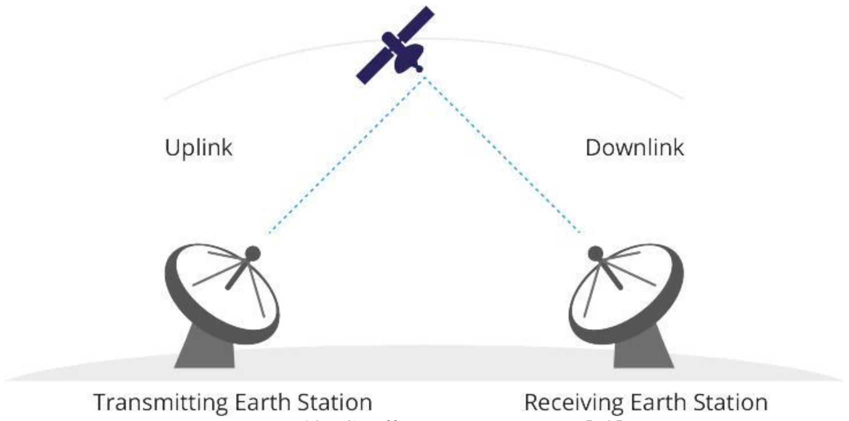


Figure 13 - Satellite communication [52]

Satellites have many shapes and sizes. They have two parts in common - an antenna and a power source. The antenna sends and receives information, often to and from Earth. The power source can be a solar panel or a battery [53].

Satellites are equipped with complex attitude and orbit control systems which, depending on the mission, may have extreme requirements regarding their accuracy. The payloads of satellites range from radar systems through optical systems to special applications such as gradiometers [54].

3.3 *Development and Verification*

Modern complex systems, such as spacecrafts, cars, airplanes and power plants are realized as a combination of hardware equipment and software for control. The Figure 14 depicts the development process of such systems:

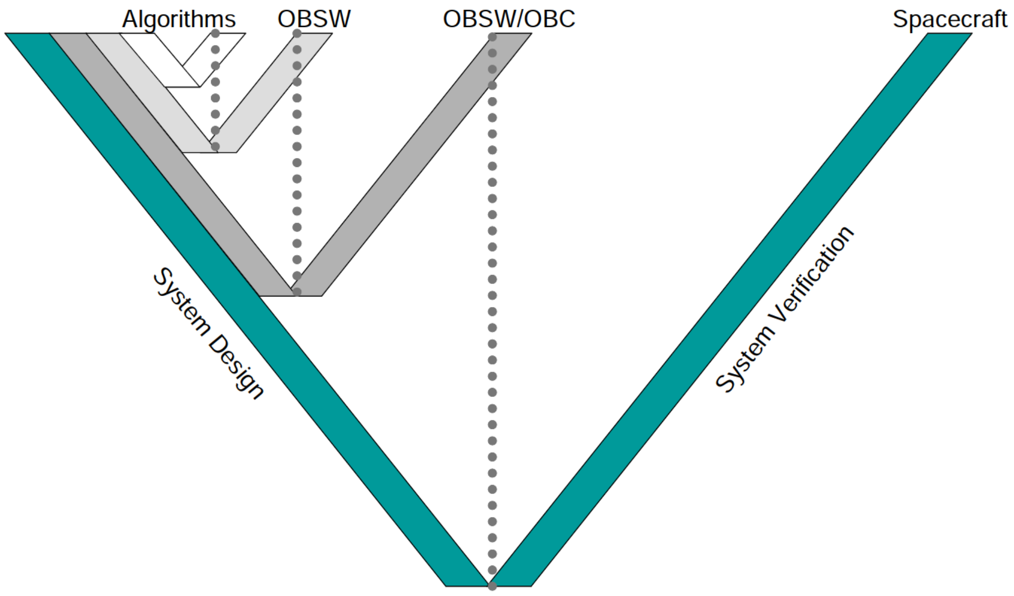


Figure 14 - Functional Design and Verification

Observing the image, we are facing a classic V-model, consisting in two main branches, design and verification. This model however, in the engineering of an entire system, breaks down to a set of interlinked V-steps [DO-178, ECSS-E-40-04, ECSS-E-40-05].

Reading the Figure 14, from right to left, the author [54] explains that to be able to verify the detailed requirements on the system (here a spacecraft) by system testing, a suitable target computer based control software has to be available. This means, HW/SW integration must be completed for this step. To verify the proper HW/SW integration, already a pre-verified control software must be available. In spacecraft engineering, this is known as on-board software. And finally, to be able to pre-verify the on-board software, for the integrated control algorithms - e.g. for attitude control - reference data must be available. An algorithm verification campaign, must be already completed at that point.

The Figure 14 furthermore points to an important fact concerning system modelling and verification infrastructures. For verification of any functionality an infrastructure is needed, independent from the level being of simplest algorithm verification up to extremely complex top-level system tests. So, in a system development, a design and test environment must be foreseen for the control algorithms, for transformation of the algorithms into on-board software code (before being integrated with target hardware), for hardware/software integration and finally, for the entire system (spacecraft) including its integrated on-board computer.

The verification concept for a spacecraft including its on-board software, as shown in Figure 14, already depicts a stepwise development principle, which also is applied in many other industries.

In an initial step, the physics of the system is modelled - in software or as a test stand - and the developed control algorithms are integrated to control the system. This type of test is called “Algorithm in the Loop”. Secondly, the algorithms are coded in software in the target language. The new control software is loaded onto the - eventually modified - test stand, again, to control the system. This type of tests is called “Software in the Loop”. The third step is to load the control software onto a representative target computer, which now controls the hybrid test stand. The final software on the target computer now has simulated system physics. This principle is called “Controller in the Loop”. The fourth and final step of system testing now aims to make the control software on the target hardware now control the real system, and no longer the test stand's system simulation. This deployment phase is called “Hardware in the Loop”, HITL.

For all these steps, the requirements documentation and design for the SUT must be written, the principal verification approaches are to be documented and finally, test plans for the verification are to be generated and results are to be collected and analysed in test reports.

This kind of simulation based system development approach requires fundamentally new workflow processes to be applied, both concerning applied technology as well as with respect to project organization and distribution of responsibilities.

In brief, what is to be managed is the integration of engineering disciplines such as mechanics/kinematics, electrics, thermal and system operations/data handling, the allocation of a simulator infrastructure responsible to the project. This role, also called “simulator architect”, is a task that manage the in-time and requirement compliant development and qualification and installation of the simulation infrastructure.

The tasks to be managed furthermore comprise the consistent application of simulators, system models, configuration databases and test procedures to all project phases. The system design, simulation and verification environment should be standardized as much as possible to reuse qualified elements in the next space project and finally, a consolidated work process is needed for the integrated development and test of satellite hardware and software, system simulator and check-out software and equipment.

It has to be kept in mind that the functionality of the simulation based test stands has to be defined, implemented and verified following an analogous process as explained for the spacecraft itself above.

3.4 *Software Verification Facility (SVF)*

In the development process of a satellite, a SVF is implemented. The programmed on-board software of the satellite is pretested on this test bench for the first time.

Considering an on-board computer (OBC) with a classical computer architecture, the on-board software (OBSW) includes the operating system of the on-board computer, the spacecraft system control code, including all control algorithms, all interface drivers for input/output interfaces between on-board computer and satellite equipment and functions necessary to receive telecommands from ground and to generate satellite telemetry for the ground station. These software elements must be tested extensively.

An SVF is equipped with a detailed model of the on-board computer. In the SVF's OBC simulation model, all components, including the microprocessor or "central processing unit" (CPU), are exactly reflected regarding their functionality. The on-board software, compiled for target CPU and periphery hardware architecture can thus be loaded directly into the SVF OBC model and can control the simulated satellite. This is also applicable for the OBC's basic I/O-system, BIOS, boot software and the operating system of the on-board computer. The SVF infrastructure thus corresponds to the "Software in the Loop" implementation step in the development process [54].

Furthermore, the SVF is not only used for on-board software tests in the scope of attitude/orbit control. The SVF should enable control and monitoring of all functions of the simulated spacecraft. For satellites, this includes Attitude and Orbital Control System (AOCS), platform and payloads - all controlled by OBSW in the loop.

In a day-by-day work, the SVF corresponds to an Eclipse integrated development environment (IDE) console that is provided by the SVF team to the validation team. This console allows the validation team to write tests in Java and run them in the same place.

This SVF, have a Java lib that contains methods that allows testers to test a specific component of the spacecraft. Sometimes, all methods needed already exist, since most of the code of satellites is reused from previous ones, or new methods and classes can be created for new components or updated components. Behind, and hidden from testers, this console, have the models. These models correspond to the simulated hardware of the satellite.

The main idea to keep from SVF is that is a satellite simulator that allows testers to write tests using methods provided by a Java lib, or create new ones, to validate the OBSW. These tests are written in Java and as we saw the documentation, a system verification test specification (SVTS) including the requirements covered, the checks and the state conditions should be created.

4 Enabling Keyword-Driven Validation

In this dissertation, the main task is to define a test frontend API that enables simple test procedures for a satellite instrument control application. As the master dissertation proposal highlights, the framework chosen to implement this API was the Robot Framework because of its capabilities to use keyword-driven approach.

As already presented, the SVF can be intended as an Eclipse IDE that contains all the models, simulated hardware and methods. These models behind the IDE are hidden, they are exclusively updated by the SVF team so, can't be used or adapted to create keywords and used by Robot Framework. The SVF acts as a black-box environment.

Since we face this black-box environment, the normal application of the Robot Framework to the project was not possible even using the framework plugin for the IDE, RED plugin. The RED plugin is modern editor based on Java IDEs (Eclipse, IntelliJ in future) to allow quick and comfortable work with Robot Framework testware [50]. The IDE is a company own IDE and it's not allowed to install or update plugins. The IDE received by the testers contains all the plugins and features needed for the tester perform the job and no changes or updates can be performed.

Facing this issue, we intend to develop a Python application, that can be used in this kind of systems and allow us to use the keyword-approach at same time. This chapter aims not only to explain how the implementation was done, showing diagrams when appropriate, but also to explain the reasoning behind the decisions that were made throughout the implementation, giving a more powerful insight of the work done in this dissertation.

4.1 *Merging the API in the Validation Process*

Before present the API in detail, we will see the validation process that is applied in the satellites field. Understanding this process, we will demonstrate what is the intention of the API and where the API will fit the process.

4.1.1 Validation Process

The validation process is a classical manual strategy that was already presented in section 2.5.1. As depicted in Figure 15, the validation process contains four main phases.

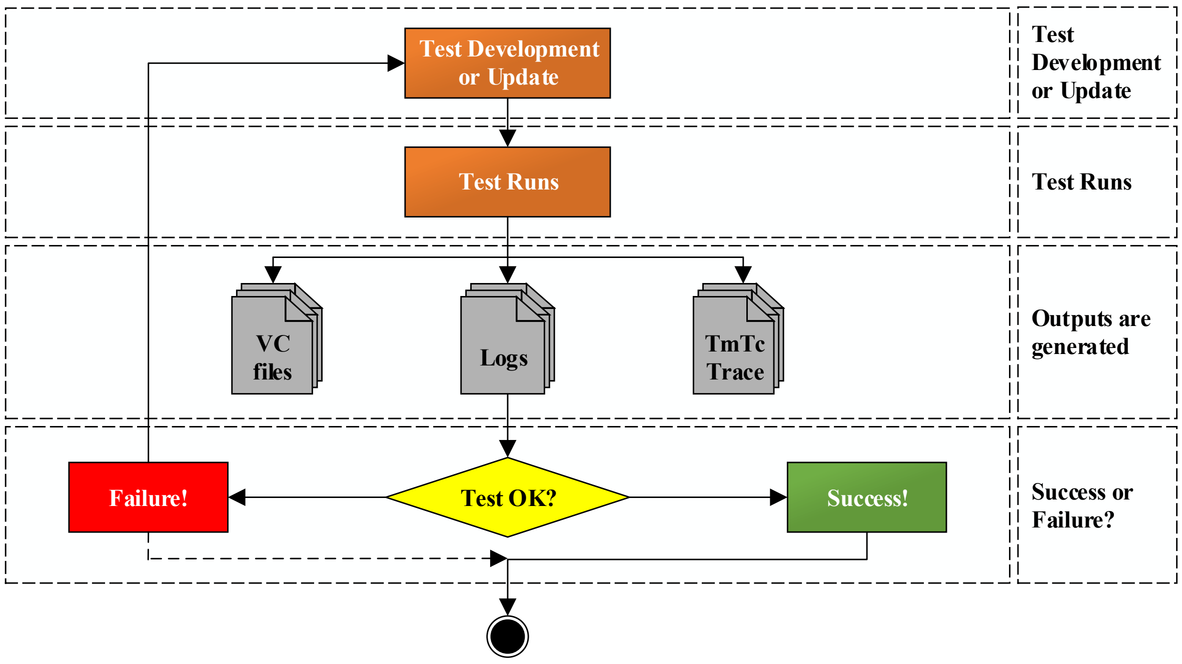


Figure 15 - Software Validation Process [55] and [56]

Reading the phases depicted in Figure 15, from the top to down, the first one corresponds to test development or update, where a Java test is developed or updated according the specification, SVTS.

In the next phase, the test run. Having the test finished or updated according the SVTS we need to run the test to collect results. The tests run directly in the SVF.

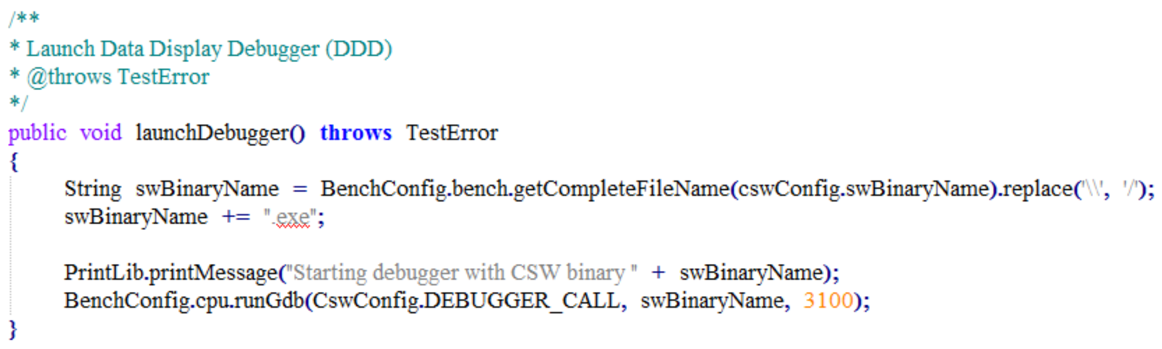
The phase after, is a phase that we collect all the files generated by the test. In this pool of files, we can find a variety of files like, logs that inform the activities performed, the expected results of each action and the evaluation between the expected and the received values. We can find also the TmTc Trace file that contains all the messages that flow during the time that the test was running. This file is very helpful for debug purposes.

Finally, the last phase is a very important one that involves a serious analysis. In this phase, one of three paths can be followed depending of the global output state of the test.

If the test is OK, the log doesn't show errors, no more updates need to be performed and the process ends. In case that the test is KO we came across to a "junction". At this stage, the tester need to analyse the cause of the error that generate a failed test. If the test is well written and its according the SVTS the problem resides in the code that we are testing. We are facing a software bug. In this case, no updates are done to the test, the test remains untouchable until a new software release is available and the process ends. In other way, if the test is KO and no software bugs are found this means that the problem resides in the test. The process starts again updating the test until we get a state that the test is OK or KO. We can obtain again a KO, but justified that the problem does not reside in the test.

4.1.2 API in Validation Process

The API provides to the tester a suitable way to develop tests using the keyword-driven approach. This API see each method, in the SVF, as a keyword. Instead of developing tests using Java language, the tester will produce tests using keywords. These keywords act like a mask for Java code, for example, the keyword *launchDebugger* can have a Java method associated like as depicted in Figure 16.



```
/**
* Launch Data Display Debugger (DDD)
* @throws TestError
*/
public void launchDebugger() throws TestError
{
      String swBinaryName = BenchConfig.bench.getCompleteFileName(cswConfig.swBinaryName).replace('\\', '/');
      swBinaryName += ".exe";

      PrintLib.printMessage("Starting debugger with CSW binary " + swBinaryName);
      BenchConfig.cpu.runGdb(CswConfig.DEBUGGER_CALL, swBinaryName, 3100);
}
```

Figure 16 - Code example for launchDebugger keyword

The API is a Python tool that allows the tester to create tests using keywords, these tests, when finished, will be translated to Java test files. With this, we are changing the way how the tests are developed, i.e., we are changing the top phase of the validation process showed in Figure 15.

Knowing this, we can see in the Figure 17 the validation process now with the developed API.

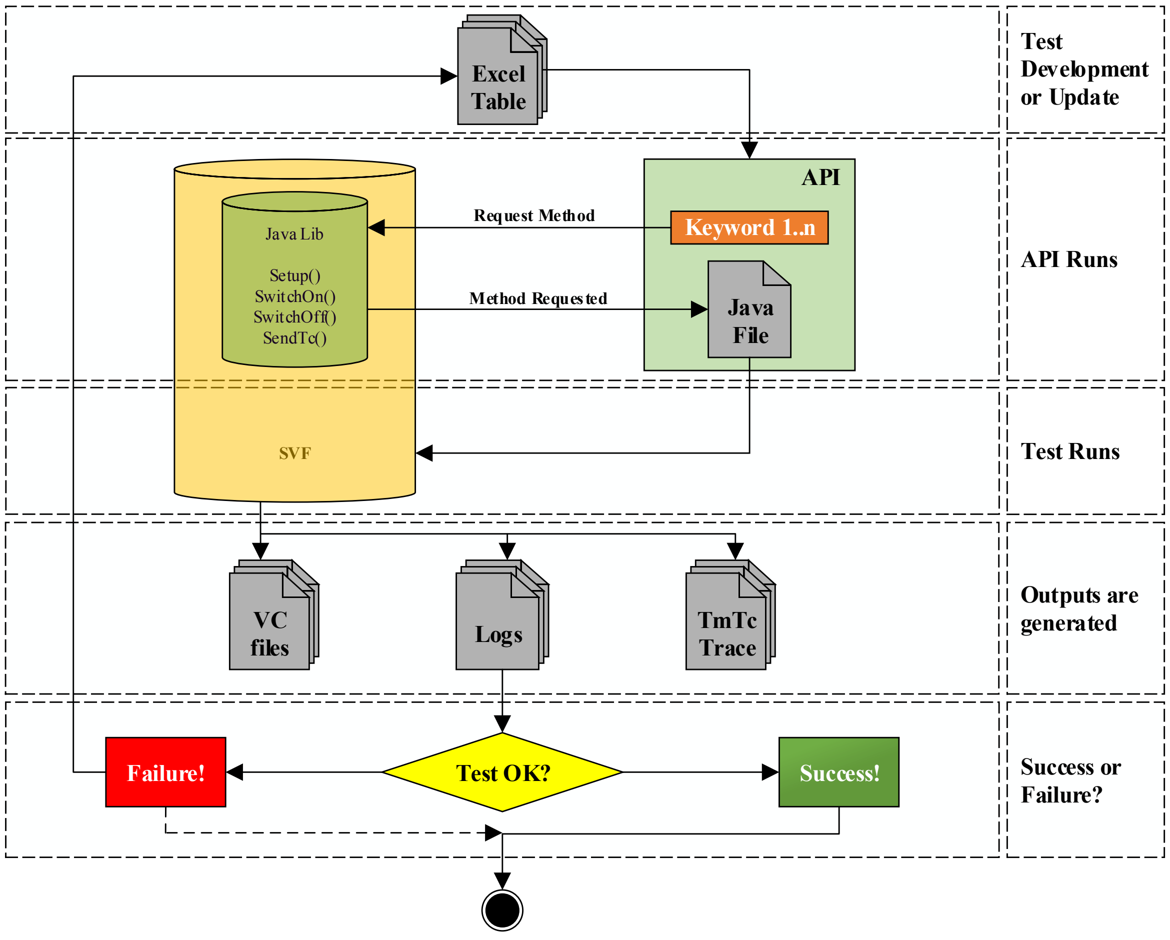


Figure 17 - Role of API in the validation process

The top validation process from Figure 15, Test Development or Update, will be substituted by two phases called Test Development or Update and API runs, as depicted in Figure 17.

The first phase will maintain the same objective, is the phase where tests are developed or updated. The difference is that when we are using the API the tests are developed not in Java but in Keywords. Instead of developing a complex Java file with code, the tester will produce a test just filing a table divided by steps and activities with keywords. To see the available keywords and functionality, the tester need to run a method, that will be introduced, at least once. At the end of this phase, an excel file will be obtained containing a table with keywords.

The second phase, is the phase where the API will translate the keyword test to Java code. Having the test finished, the API will translate the keyword test in a Java test. At the end of this phase, we will obtain a Java file that can run in the SVF.

After this, and looking to Figure 15 and 17, the next phases remain the same. Once that we have a Java file, it will run and generate results that will be then scrutinized.

4.2 *Implementing a test frontend API*

The main idea of the frontend API follows the concept of keyword-driven approach that create functions forming the function library and call them as keywords in the tests in each keyword is associated with separated functions.

As we saw, in the available SVF, we already have functions that perform actions, the Java lib, or new functions can be created. Having the functions, one part of the concept is achieved. Apart from this, the developed API shall provide to the testers keywords that are directly linked to the functions.

Furthermore, the API, will act as a code translator. First, we will have a tabular test that contain keywords, that should exist in the code as a function, and then, this tabular test will be translated to a Java code.

Having the main idea and the two main steps that form the API a detailed description will be presented.

For the first step, the keyword and the link to the Java function needs to be created. To achieve this, the keywords will correspond directedly to the name of the function, i.e., for the method in Java "*startPCDU*" the keyword is "*startPCDU*". The name of the Java method and the keyword are exactly the same, this allows us to have sure that when we use a keyword we can easily find the correspondent Java method in the SVF. After this, we can easily create tabular tests using keywords.

For the second point, in order to translate the tabular test in Java code, the API will have an algorithm that for each keyword a method call will be added to the Java test and, of course, the correspondent package, constructers and objects that the Java language obligates will be added too.

As depicted in Figure 18, the API includes two distinct methods that do the two distinguish jobs presented before.

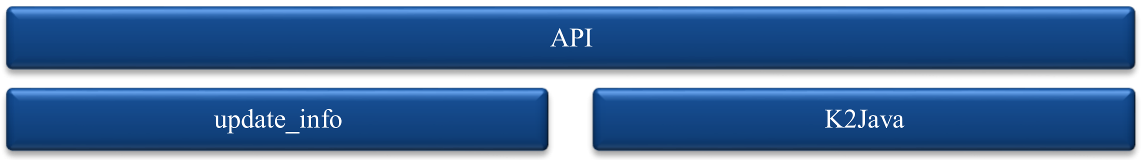


Figure 18 - API methods

The first method, *update_info*, is responsible for generate documentation to the tester. This document, more properly a HTML page, will be the user manual that the tester need to use to develop tests. In the page, the tester can find the keywords available, the functionality associated to each one and more information about the package that the keyword belongs and a link to the Java code.

The second method, *K2Java*, is the "automated tool" that enables the translation of the tabular test to Java test.

In order to conduct a proper presentation of the methods presented above, the next section will follow a specific line. First, we will present the description and objectives of the first method, *update_info*. Knowing the objectives, we will summarize the requirements that this method need to obey. Finally, having all the information, the developed solution will be presented, including the description of each function and a flowchart to better understand the work done. Secondly and finally, the second method, *K2Java*, will be presented following the same line as before.

4.2.1 *Update_info* Method

As we saw above, the *update_info* method comprehends a series of tasks that provide the keywords to the tester, as already said the keyword correspond to the name of the Java method. This allows us to make sure that the tester uses keywords that have code (Java method) linked to them.

Knowing this, we can see the *update_info* method as an algorithm that iterates throw all Java lib (methods provided by the SVF) and returns to the tester a list of keywords that, actually, are the names of all methods in this lib.

Knowing the objective of the functionality we can present the requirements needed to obtain this end.

Requirement A.1: The *update_info* shall iterates all Java lib (in the SVF) and collect the names of Java methods. Only methods shall be considered.

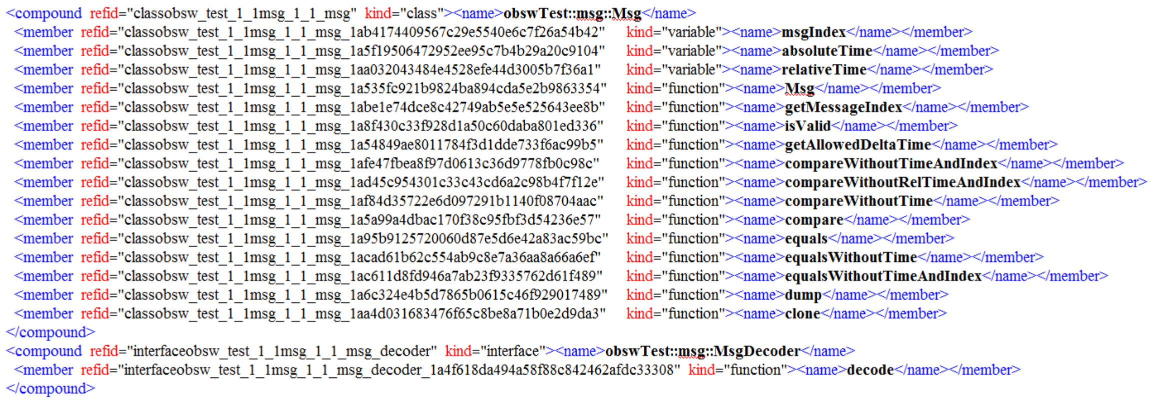
Requirement A.2: For each Java method name the correspondent package that this method belongs shall be retrieved.

The idea is to cover all Java lib, in the SVF, and collect all the names of methods present. For this purpose, we use a documentation generator instead of creating a script from scratch. Some documentation generators can read Java code and act as a parser.

In our solution, the documentation generator used was *Doxygen*. *Doxygen* can help the user in three ways [55]. First, it can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in *Latex*) from a set of documented source files. Second, can be configured to extract the code structure from undocumented source files. *Doxygen* can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically. Finally, this tool can be used for creating normal documentation.

As we saw, *Doxygen* can provide us the information requested by *update_info* method, the methods that are present in the source files. In our solution, a *Doxygen* configuration file was created to extract only the methods and correspondent package automatically. The information is extracted to a XML file (*index.XML*) and saved in the *Doxygen* output folder automatically when *Doxygen* runs.

As depicted in Figure 19, the XML file generated by *Doxygen* follow a specific structure. Each Java package corresponds to a <*compound*> and the variables, constructors or functions to <*member*>.



```xml
<compound refid="classobsw_test_1_1msg_1_1_msg" kind="class"><name>obswTest::msg::Msg</name>
  <member refid="classobsw_test_1_1msg_1_1_msg_1ab4174409567c29e5540e6c7f26a54b42"    kind="variable"><name>msgIndex</name></member>
  <member refid="classobsw_test_1_1msg_1_1_msg_1a5f19506472952ee95c7b4b29a20c9104"    kind="variable"><name>absoluteTime</name></member>
  <member refid="classobsw_test_1_1msg_1_1_msg_1aa032043484e4528efe44d3005b7f36a1"    kind="variable"><name>relativeTime</name></member>
  <member refid="classobsw_test_1_1msg_1_1_msg_1a535fc921b9824ba894cda5e2b9863354"    kind="function"><name>Msg</name></member>
  <member refid="classobsw_test_1_1msg_1_1_msg_1abe1e74dce8c42749ab5e5e525643ee8b"    kind="function"><name>getMessageIndex</name></member>
  <member refid="classobsw_test_1_1msg_1_1_msg_1a8f430c33f928d1a50c60daba801ed336"    kind="function"><name>isValid</name></member>
  <member refid="classobsw_test_1_1msg_1_1_msg_1a54849ae8011784f3d1dde733f6ac99b5"    kind="function"><name>getAllowedDeltaTime</name></member>
  <member refid="classobsw_test_1_1msg_1_1_msg_1afe47fbea8f97d0613c36d9778fb0c98c"    kind="function"><name>compareWithoutTimeAndIndex</name></member>
  <member refid="classobsw_test_1_1msg_1_1_msg_1ad45c954301c33c43cd6a2c98b4f7f12e"    kind="function"><name>compareWithoutRelTimeAndIndex</name></member>
  <member refid="classobsw_test_1_1msg_1_1_msg_1af84d35722e6d097291b1140f08704aac"    kind="function"><name>compareWithoutTime</name></member>
  <member refid="classobsw_test_1_1msg_1_1_msg_1a5a99a4dbac170f38c95fbf3d54236e57"    kind="function"><name>compare</name></member>
  <member refid="classobsw_test_1_1msg_1_1_msg_1a95b9125720060d87e5d6e42a83ac59bc"    kind="function"><name>equals</name></member>
  <member refid="classobsw_test_1_1msg_1_1_msg_1acad61b62c554ab9c8e7a36aa8a66a6ef"    kind="function"><name>equalsWithoutTime</name></member>
  <member refid="classobsw_test_1_1msg_1_1_msg_1ac611d8fd946a7ab23f9335762d61f489"    kind="function"><name>equalsWithoutTimeAndIndex</name></member>
  <member refid="classobsw_test_1_1msg_1_1_msg_1a6c324e4b5d7865b0615c46f929017489"    kind="function"><name>dump</name></member>
  <member refid="classobsw_test_1_1msg_1_1_msg_1aa4d031683476f65c8be8a71b0e2d9da3"    kind="function"><name>clone</name></member>
</compound>
<compound refid="interfaceobsw_test_1_1msg_1_1_msg_decoder" kind="interface"><name>obswTest::msg::MsgDecoder</name>
  <member refid="interfaceobsw_test_1_1msg_1_1_msg_decoder_1a4f618da494a58f88c842462afdc33308"  kind="function"><name>decode</name></member>
</compound>
```

Figure 19 - Sample of XML file generated by Doxygen

Analysing the Figure 19, we can observe that *Doxygen* provides all information needed in a structured way. The *Doxygen*, for each class, provides us the functions in this class, for example, for class *obswTest::msg::Msg* we have *Msg, getMessageIndex, isValid* and so on. Of course, we can see that not only functions are collected, we can see *msgIndex*, *absoluteTime* and *relativeTime* variables too.

Understanding the structure of the XML file, a function was developed to collect only the information needed, i.e., the methods (functions) and the package of each one. Using the Figure 19 as an example, we can describe the way how to collect the information. The functionality iterates the *index.XML* file extracting the methods, this means collects the lines that *kind = "function"* and the correspondent package, *kind = "class"*, of each one.

Before present the architecture in detail, we need to remind that the *update_info* method shall collect the keywords from Java but also present the information to the tester in a proper way. As we will see, we will use the *Doxygen* capabilities once again to present the information. Although, what this time, when the keywords are collected, two files are created. The first file, an txt file, will save a list of keywords and the correspondent packages, this file is called *key_database*. The second one, correspond to a html file with code to build a table with keywords. In the next sections, we will observe the importance of these two files and a proper explanation will be presented.

Once we explain the overall *update_info* method functionality, we will refine the explanation and present the functions that compose this method. The functions will be described in detail and by order that they will be called to perform the expected functionality.

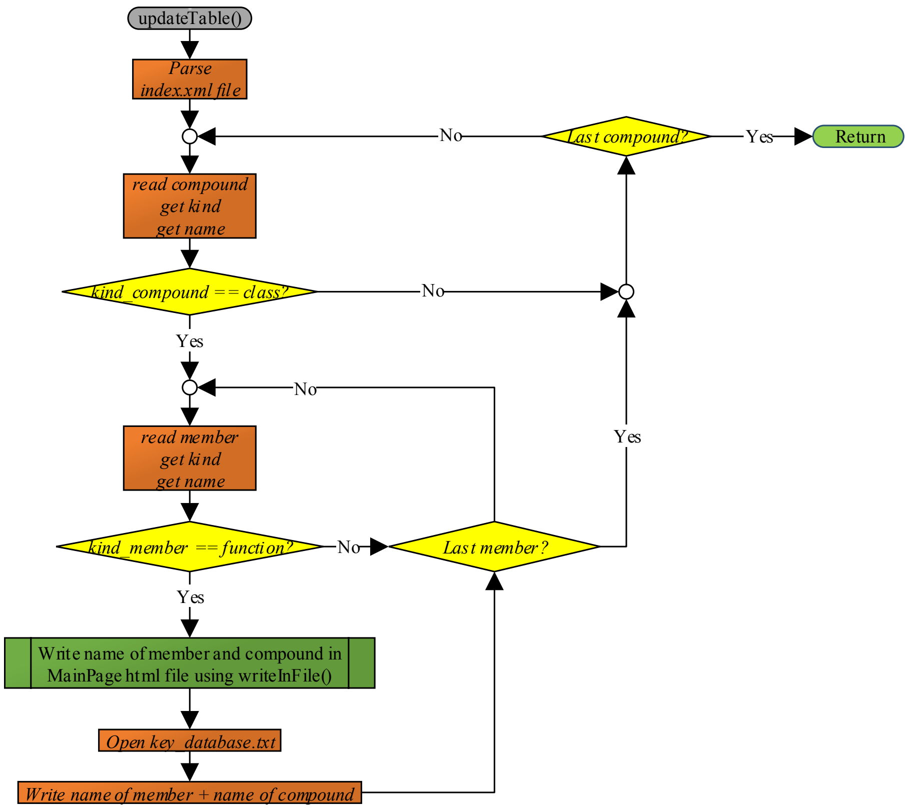


Figure 20 - UpdateTable function

In Figure 20, the *updateTable* function is presented. This function makes part of the *update_info* functionality and is responsible for retrieve the keywords from *Doxygen*.

To better understand the diagram, we will present an example having as input the Figure 19. The algorithm starts using some Python functions to parse the XML file. With these functions we can read the first *<compound>* getting its kind and name, for example, *kind = "class"* and *name = "obswTest::msg::Msg"*. If the kind of the compound is a class, the process continues to read the members inside. If not, jump to the following compound if there is any. In case that there's no more compounds the process ends.

In this example, the process continues to read the members since the *compound* is *"class"*. Once again, we will check in the same way the kind of the member. If the member isn't a function the algorithm jumps to the next member (if any) or can pass to the next compound (if any).

The first member of *"obswTest::msg::Msg"* have a kind equals to "*variable*", it's not a function. The algorithm jumps to the next member and check the kind again. Once again, the kind isn't the expected. The process iterates throw all members until that finds a function with name equals to *"Msg"*.

Having a member that fill all requirements, the name and the compound that it belongs, are written in both files presented before, the *key_database* and the file that contain html code, *MainPage* file. As depicted in Figure 20, the *key_database* is updated using simply Python functions. But, for *MainPage* html file, a new function was mentioned. The explanation of this new function, *writeInFile*, is simply write in files. As we said, the

MainPage html file is a file with html code to build a table so, every time a new keyword need to be added, we need to find the last line of the table and insert a new one. This function is a general function that was used several times to write strings in files.

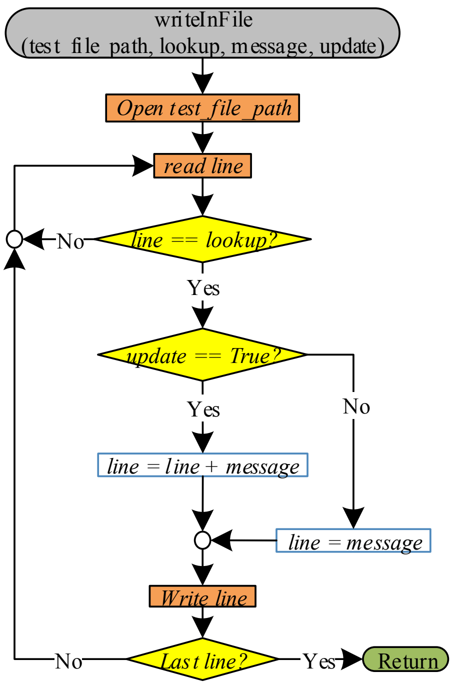


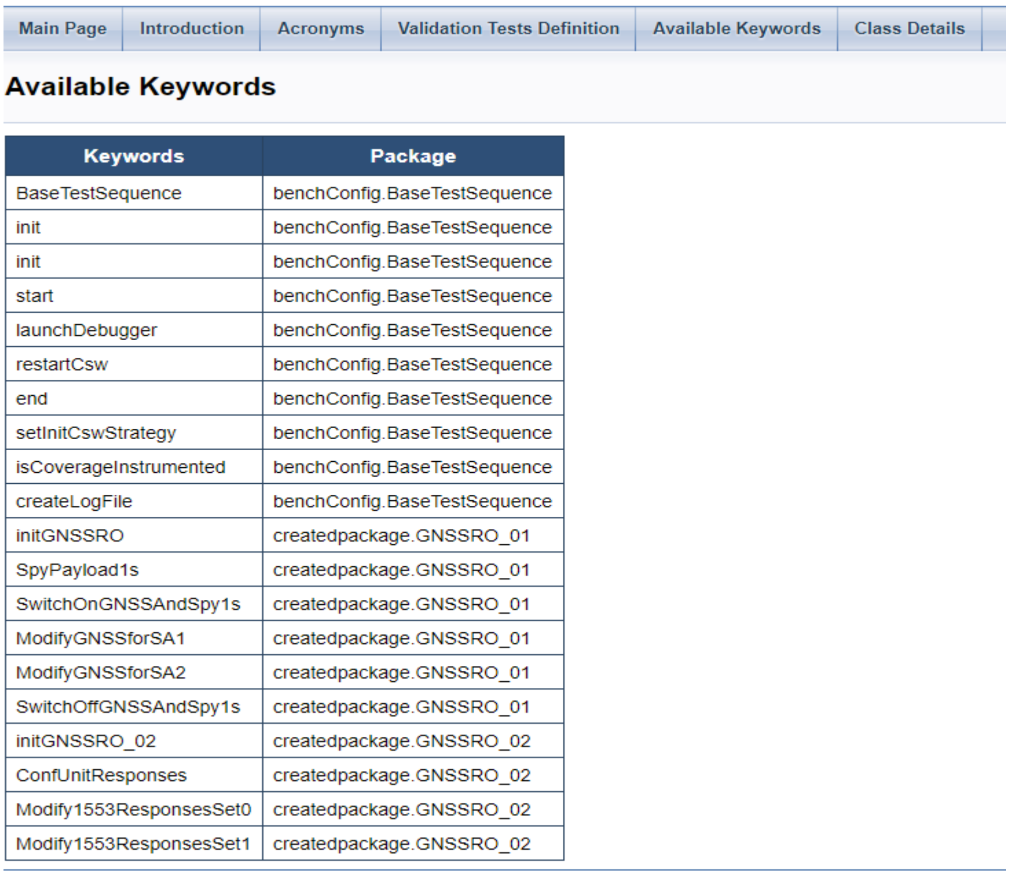
Figure 21 - WriteInFile function

As depicted in Figure 21, the *WriteInFile* function is responsible for write strings (message) in a specific file passed as argument. The function can overwrite lines or simply add content after a specific line. For the current case, the function was used to add lines in the html table code in *MainPage* file.

With the example, we can understand the information generated by *Doxygen*, mainly the structured way how packages and functions are collected and the develop approach that retrieve the information important for the tester.

Since the information was collected we need to present it to testers in a proper way. As already said, *Doxygen* have some capabilities that allows us to generate html pages. *Doxygen* generates a html page by default with classes, functions, variables, links to the code, etc. Our intention was present to the tester an html page that present a table with keyword and description. With this, the tester only need to check the table and search for the suitable keyword that perform the check that he/she needs. The idea is to use the html page as a manual to develop tests.

The *Doxygen* generates automatically an html page with a few standard tabs and allows the user to add new ones but, they need to be written in html. The explanation about why we create a html file when the keywords are collected can now be presented. As depicted in Figure 20, we saw that every time that a new keyword (function) is found, is written in a html file that contain code to build a table. This file is then used by *Doxygen* that creates a new tab as depicted in Figure 22. We call to this new tab "*Available keywords*".



Available Keywords

Keywords	Package
BaseTestSequence	benchConfig.BaseTestSequence
init	benchConfig.BaseTestSequence
init	benchConfig.BaseTestSequence
start	benchConfig.BaseTestSequence
launchDebugger	benchConfig.BaseTestSequence
restartCsw	benchConfig.BaseTestSequence
end	benchConfig.BaseTestSequence
setInitCswStrategy	benchConfig.BaseTestSequence
isCoverageInstrumented	benchConfig.BaseTestSequence
createLogFile	benchConfig.BaseTestSequence
initGNSSRO	createdpackage.GNSSRO_01
SpyPayload1s	createdpackage.GNSSRO_01
SwitchOnGNSSAndSpy1s	createdpackage.GNSSRO_01
ModifyGNSSforSA1	createdpackage.GNSSRO_01
ModifyGNSSforSA2	createdpackage.GNSSRO_01
SwitchOffGNSSAndSpy1s	createdpackage.GNSSRO_01
initGNSSRO_02	createdpackage.GNSSRO_02
ConfUnitResponses	createdpackage.GNSSRO_02
Modify1553ResponsesSet0	createdpackage.GNSSRO_02
Modify1553ResponsesSet1	createdpackage.GNSSRO_02

Figure 22 - Available Keywords Tab Doxygen

As depicted in Figure 22, the table only shows keywords and package, the description of the keyword isn't present. This is explained because *Doxygen* only provide us files like as depicted in Figure 16. *Doxygen* collects the comment in the Java code yes, that correspond to the description of the keyword, but this information only appears in "Class Details" tab. There's no way, at this time, to build a table with keywords and description.

With the information above, we describe the developed solution to aim the *update_info* method goals, this means, run *Doxygen*, collect the function names from Java (keywords) and present them in a proper way to the testers. We verified a difficulty, present a table with keywords and description to the tester, a difficulty that we can't overcome since we are using *Doxygen*.

Another difficulty was found when we run *update_info* functionality several times. When we run the functionality, *Doxygen* runs and all the data is collected (keywords). We observe that the new tab doesn't appear in the html file and every time that we run *Doxygen* again the database doesn't reset. We are just putting all the same stuff over and over in *key_database* and *MainPage* file.

To overcome this issue, we came across with the sequence depicted in Figure 23 with a new function depicted in Figure 24. The *stateDoxygen* was created to overcome the gap in *update_info* method.

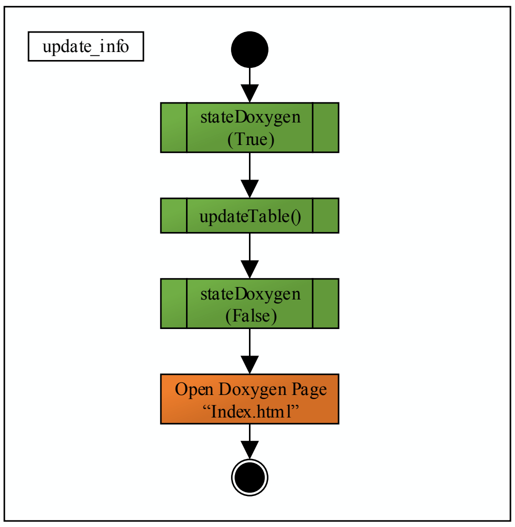


Figure 23 - update_info method internal functions

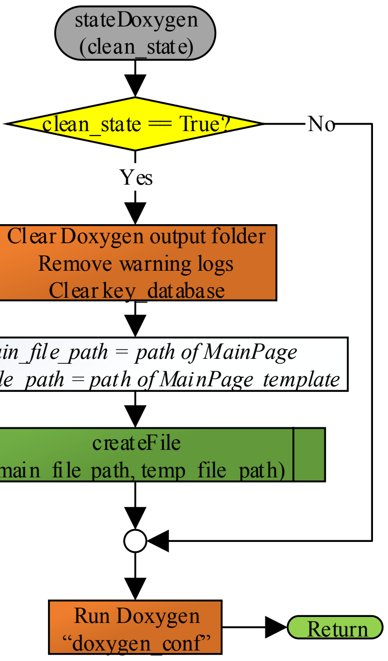

stateDoxygen (clean_state)
clean_state == True?
No
Yes
Clear Doxygen output folder
Remove warning logs
Clear key_database
main_file_path = path of MainPage
temp_file_path = path of MainPage_template
createFile (main_file_path, temp_file_path)
Run Doxygen "doxygen_conf"
Return

Figure 24 - stateDoxygen function

The *stateDoxygen* perform two actions according an input passed as argument. As depicted in Figure 24, we can observe that if this input is True, the *Doxygen* output folders are removed and the *key_database* file is cleaned. Otherwise, these steps are skipped and, as the same for the first scenario, *Doxygen* runs and the *MainPage* is created. The *MainPage* is a copy from a template since, this file complains same html code to build the table. To create the *MainPage* file a new function is used, the *createFile* function as depicted in Figure 25.

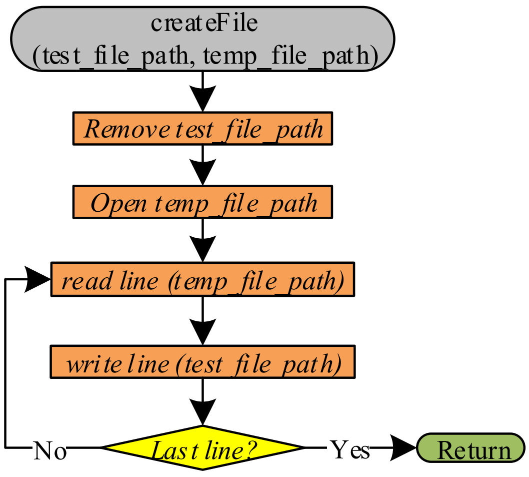

createFile
(test_file_path, temp_file_path)
Remove test_file_path
Open temp_file_path
read line (temp_file_path)
write line (test_file_path)
No
Last line?
Yes
Return

Figure 25 - createFile function

This function just copies the content of a given template to a new file, in this case called *MainPage*.

Understanding the functions that make part of the *update_info* method, we can explain the overall process as depicted in Figure 23. The process starts with function *stateDoxygen* with input set to True, this means, all the output folders, *key_database* and *MainPage* are cleaned, a new *MainPage* file is created and *Doxygen* runs. We are in a clean state.

After this first step, the *updateTable* runs. The keywords and packages are collected from XML file generated by *Doxygen*, the *key_database* and *MainPage* are fulfilled. At this point, we have two files that correspond 100% to the Java lib.

Once again, the *stateDoxygen* runs but, at this time the input is set to False so, only *Doxygen* runs again. This step is extremely important because, *Doxygen* will run with intend to update the tabs to be showed. Since the *MainPage* is already created, *Doxygen* will be able to display the "*Available Keywords*" table.

Finally, the *Doxygen* html pops-up showing all the tabs and keywords found in the Java lib.

4.2.2 *K2Java* Method

As already said, the *K2Java* method is the second method that integrates the API. This method is responsible for the translation of the keywords test to Java code.

In order to create a standard process to develop the keywords table test and to make the translation easier we set up a table that shall be used to develop the tests. With this table, like the table depicted in Table 1, we can standardize how the tests are developed and the translation.

As depicted in Table 1, the keyword tests are developed using a simple table.

Procedure				
Step / Activity	1	2	3	4
1	sendTC	setValue	activateMM	▨
2	wait	sendTC	▨	▨
3	checkTC	checkTC	▨	▨

Table 1 - Example of a tabular test

As we can see, the process is simple, each keyword in the table corresponds to the step/activity in cause. Having a tabular test like Table 1, the *K2Java* method shall be able to translate it into Java code.

Similarly, to *update_info* method, the *K2Java* method need to obey a few requirements in order to do what is expected and reach all objectives.

The following requirements complains characteristics of safe run of the application, direct connection to syntax rules of Java and of course, requirements that ensure that the functionality will do what is expected that is, translate tables of tests, like Table 1, in a Java test.

Requirement B.1: The *K2Java* method shall ensure that tester uses keywords linked to a Java method, i.e., the keywords that are present in the tabular test have a Java method behind.

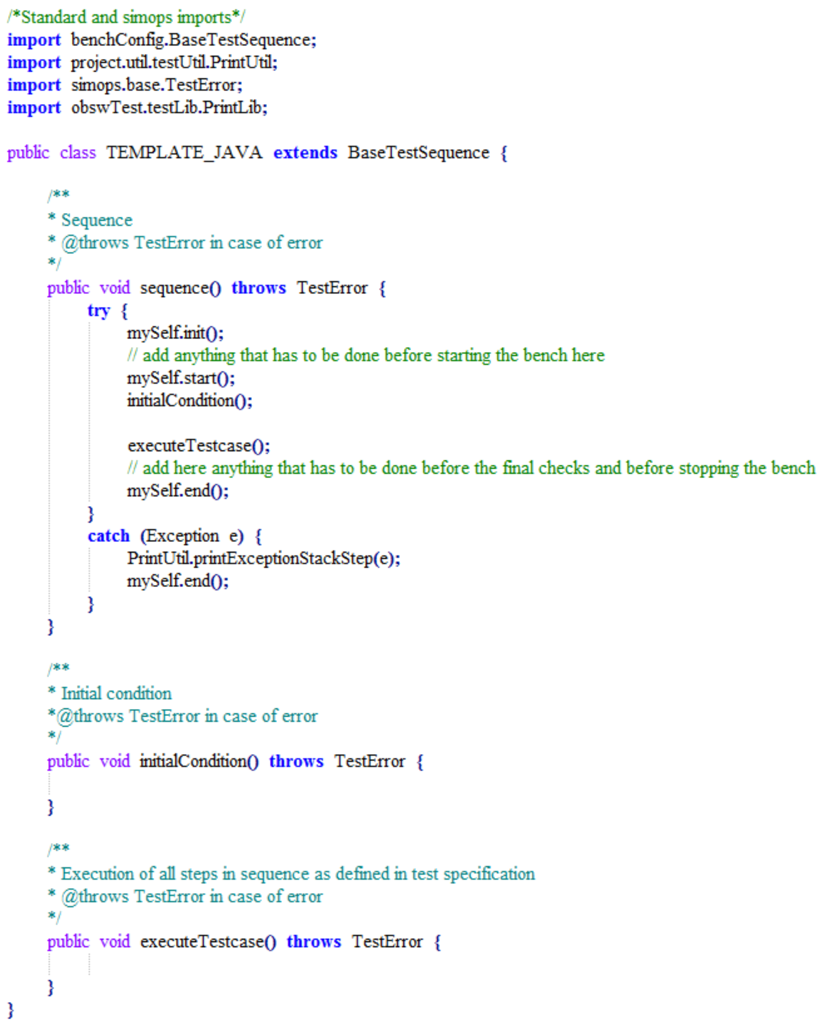
Requirement B.2: The Java tests shall be developed using a standard template that contain the structure of the test.

Requirement B.3: When a method from a particular package is added to the Java test, the constructors and objects shall be created.

Requirement B.4: If an import package, constructor and/or object was already added to the Java test this shall not be duplicated.

The requirement B.1 ensure that the translation only occurs when the keywords used exist in the Java lib. With this requirement, we can prevent future problems, for example, considering that the tabular test use a keyword that doesn't correspond to a method this will generate a syntax error in the Java code. The issue will be only discovered if the tester looks inside the Java test. This isn't the idea when using the API. The API shall abstract the Java code, i.e., the tester doesn't need to see or understand the Java code. Of course, this is the best-case scenario and as we will see this isn't achievable but, we can decrease the probability that tester need to look into the Java code.

For requirement B.2, this requirement was obtained by the experience in projects like these, validation of spacecraft systems. As observed in several projects, the Java test files for a satellite have the same structure, this means that all of them have an "*initialCondition*" method and "*executeTestcase*" method where the "real test is done". This information is very relevant because help us to use a predefined Java template that will be filled with keywords (methods calls) and the correspondent import. The Java template used is depicted in Figure 26.



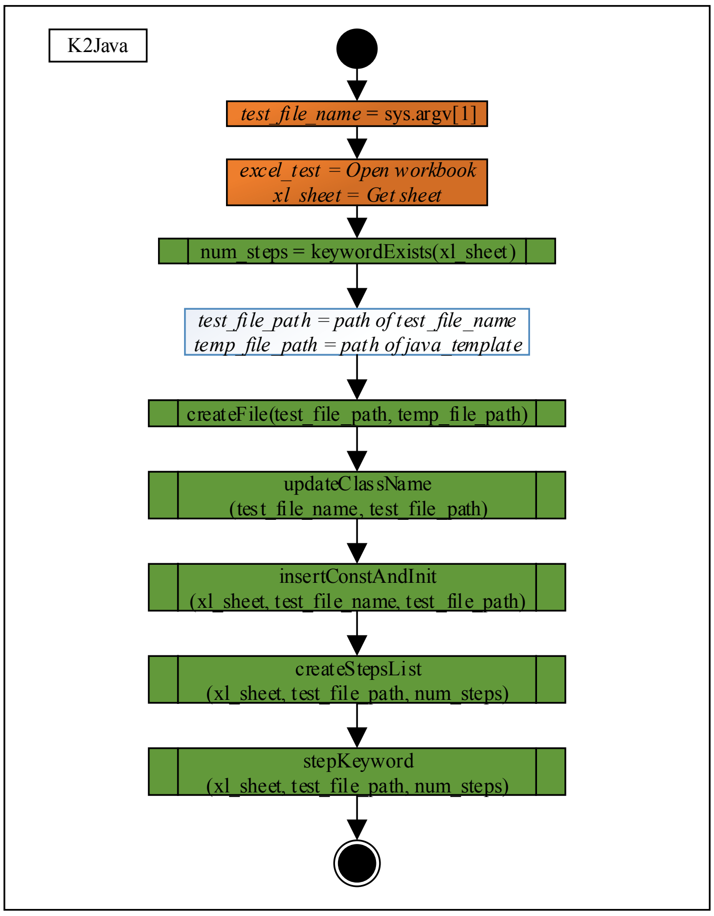
```java
/*Standard and simops imports*/
import benchConfig.BaseTestSequence;
import project.util.testUtil.PrintUtil;
import simops.base.TestError;
import obswTest.testLib.PrintLib;

public class TEMPLATE_JAVA extends BaseTestSequence {

    /**
    * Sequence
    * @throws TestError in case of error
    */
    public void sequence() throws TestError {
        try {
            mySelf.init();
            // add anything that has to be done before starting the bench here
            mySelf.start();
            initialCondition();

            executeTestcase();
            // add here anything that has to be done before the final checks and before stopping the bench
            mySelf.end();
        }
        catch (Exception e) {
            PrintUtil.printExceptionStackStep(e);
            mySelf.end();
        }
    }

    /**
    * Initial condition
    *@throws TestError in case of error
    */
    public void initialCondition() throws TestError {

    }

    /**
    * Execution of all steps in sequence as defined in test specification
    * @throws TestError in case of error
    */
    public void executeTestcase() throws TestError {

    }
}
```

Figure 26 - Java test structure

Regarding the requirements B.3 and B.4, these requirements are imposed by Java language itself.

The purpose of *K2Java* method is to translate the tabular keyword test, developed by testers, in a Java test that can run in the SVF.

Knowing the requirements and the Java structure used, we develop the following sequence as depicted in Figure 27. We will explain each function making a link with the requirements and showing what is added to the Java structure depicted in Figure 26.



K2Java
test_file_name = sys.argv[1]
excel_test = Open workbook
xl_sheet = Get sheet
num_steps = keywordExists(xl_sheet)
test_file_path = path of test_file_name
temp_file_path = path of java_template
createFile(test_file_path, temp_file_path)
updateClassName (test_file_name, test_file_path)
insertConstAndInit (xl_sheet, test_file_name, test_file_path)
createStepsList (xl_sheet, test_file_path, num_steps)
stepKeyword (xl_sheet, test_file_path, num_steps)

Figure 27 - K2Java method

As depicted in Figure 27, the *K2Java* method starts, using Python functions to access the excel file and sheet that contain the tabular test that will be translated.

Having the sheet that contain the table, the functionality deals directedly the requirement B.1 presented before. Since we need to be sure that the keywords present in the tabular test exist in the code and, at this time, the *update_info* already provide a list of available keywords (*key_database*) the process just iterate all keywords in the tabular test and check if they exist against *key_database*. The function responsible for this process is *keywordExists* function. The internal structure of this function is depicted in Figure 28.

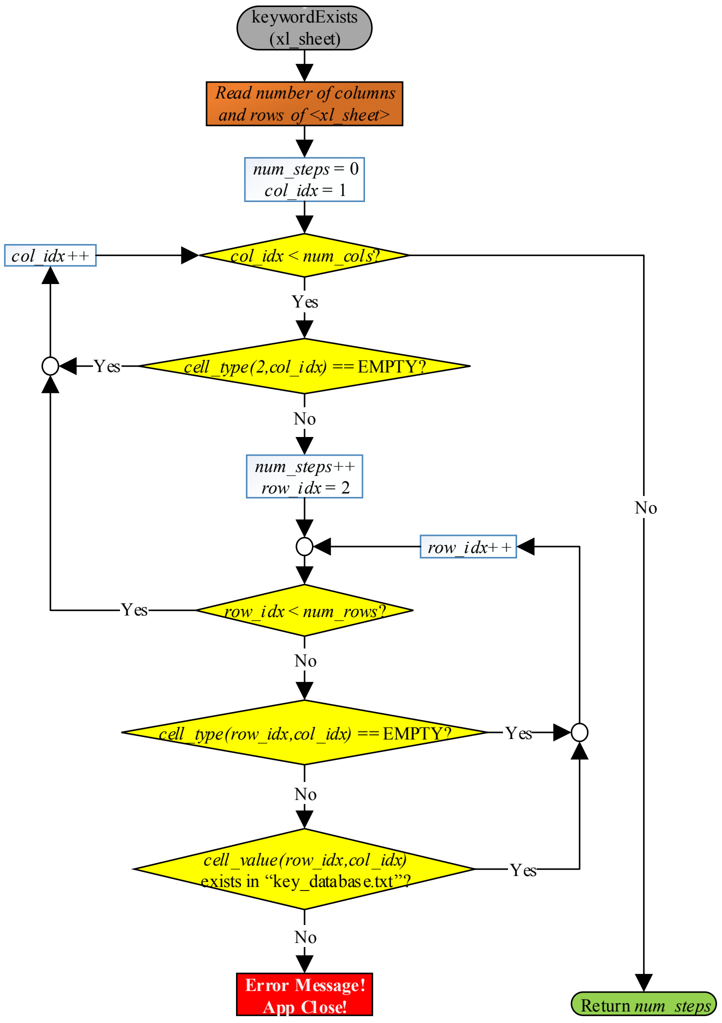

keywordExists (xl_sheet)
Read number of columns and rows of <xl_sheet>
num_steps = 0
col_idx = 1
col_idx < num_cols?
col_idx++
Yes
cell_type(2,col_idx) == EMPTY?
Yes
No
num_steps++
row_idx = 2
row_idx++
No
row_idx < num_rows?
Yes
No
cell_type(row_idx,col_idx) == EMPTY?
Yes
No
cell_value(row_idx,col_idx) exists in "key_database.txt"?
Yes
No
Error Message! App Close!
Return num_steps

Figure 28 - keywordExists function

As depicted in Figure 28, *keywordExists* function has two main objectives. First, it will act as a "keyword checker", the function iterates all columns and rows in the keyword table, like Table 1, and check if the keyword exists in the *key_database* file. The keywords are case sensitive. If a keyword doesn't exist, a message pups-up informing the tester which keyword generates the error.

Second, and since the functionality iterates all steps (columns with keywords), it will return the number of steps that the tabular test contains. We will see in the next sections why this information is so relevant.

When the function ends and, if, no error was generated, the functionality continues and create a new Java test with the name of the test in question, as depicted in Figure 25, using *createFile* function.

The function creates a copy from the Java template, depicted in Figure 26.

Having sure that all keywords exist and having a Java test for the new test, the functionality starts to update and add information to the Java code, the Java file depicted in Figure 26.

The first modification to be done in the Java file is the class name. This modification is done by *updateClassName* function as depicted in Figure 29.

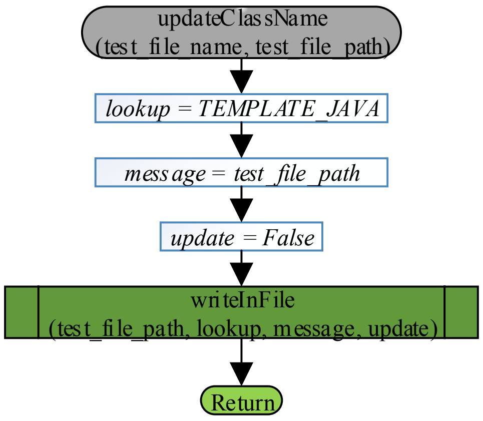
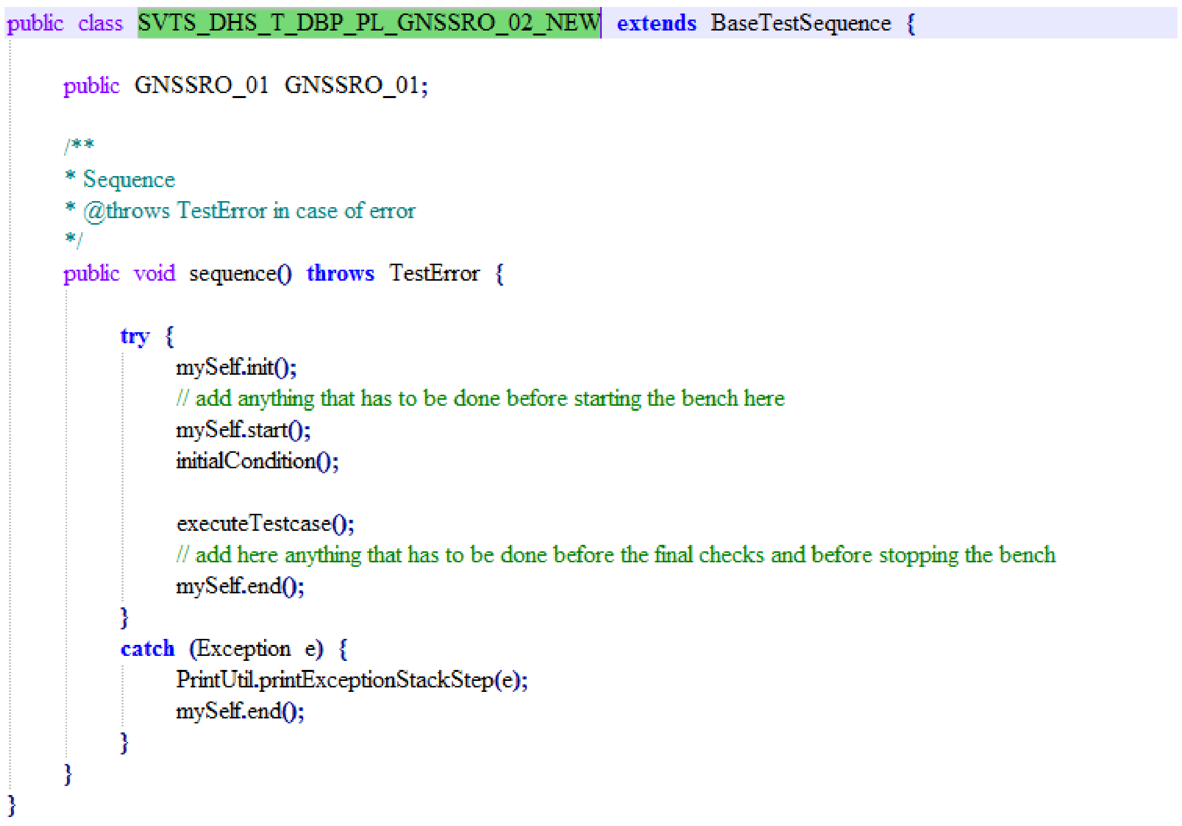


Figure 29 - updateClassName function

As depicted in Figure 29, the function builds a lookup that corresponds to the class declaration in the Java template file, build a message that is the new class declaration according with the name of the current test and finally, since in this case the line should be overwritten, call the *writeInFile* function, Figure 21, with update parameter set to False.

For example, for SVTS_DHS_T_DBP_PL_GNSSRO_02_NEW test, the Java file will be updated by *updateClassName* function and obtain the Figure 30.



```
public class SVTS_DHS_T_DBP_PL_GNSSRO_02_NEW extends BaseTestSequence {

    public GNSSRO_01 GNSSRO_01;

    /**
    * Sequence
    * @throws TestError in case of error
    */
    public void sequence() throws TestError {

        try {
            mySelf.init();
            // add anything that has to be done before starting the bench here
            mySelf.start();
            initialCondition();

            executeTestcase();
            // add here anything that has to be done before the final checks and before stopping the bench
            mySelf.end();
        }
        catch (Exception e) {
            PrintUtil.printExceptionStackStep(e);
            mySelf.end();
        }
    }
}
```

Figure 30 - Class name updated

When the *updateClassName* function finish, there's more information to be added to the Java file as the requirement B.3 obligates.

The *insertConstAndInit*, depicted in Figure 31, is responsible for extract the class of each keyword present in the tabular test. This functionality extracts all the information needed to insert the constructors, objects and initialization methods done by *createConstructor* functionality.

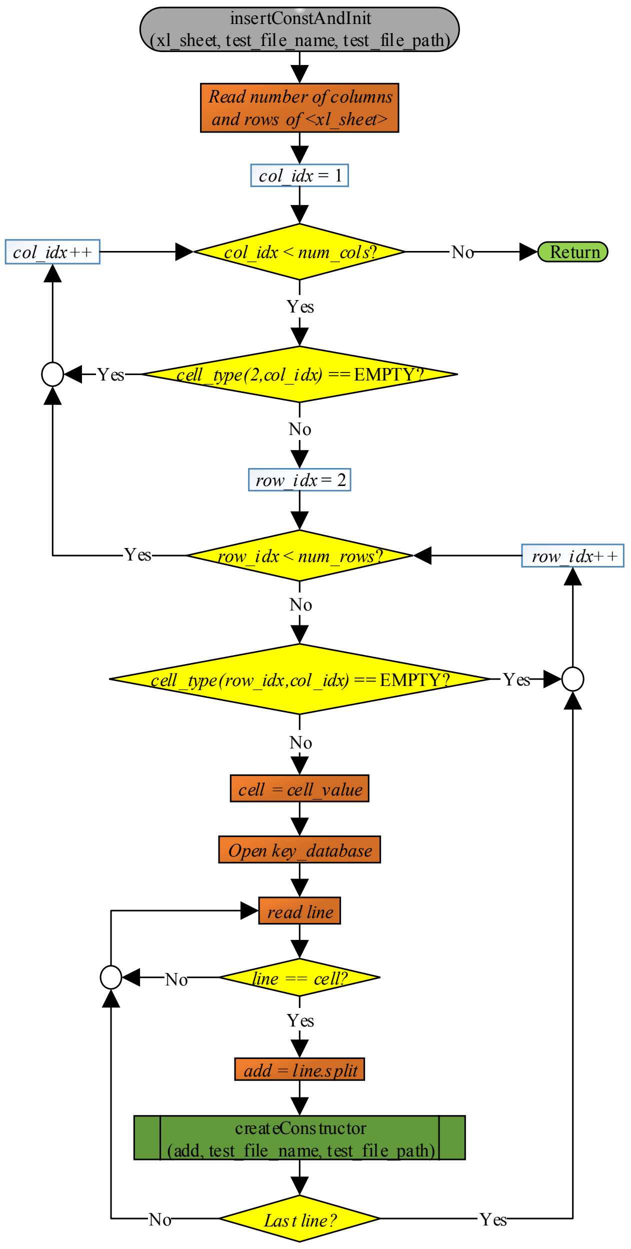


Figure 31 - insertConstAndInit function

The functionality iterates throw all keywords in the tabular test and for each one, extract from keyword database the package that this keyword belongs.

As depicted in Figure 32, the keyword database will return the package that contains the class.

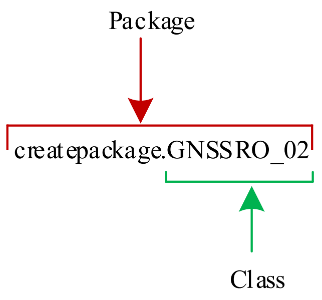


Figure 32 - Location of class name in the package

Having the extracted information, the *createConstructor* function will construct the strings to be written by *writeConstructor* function.

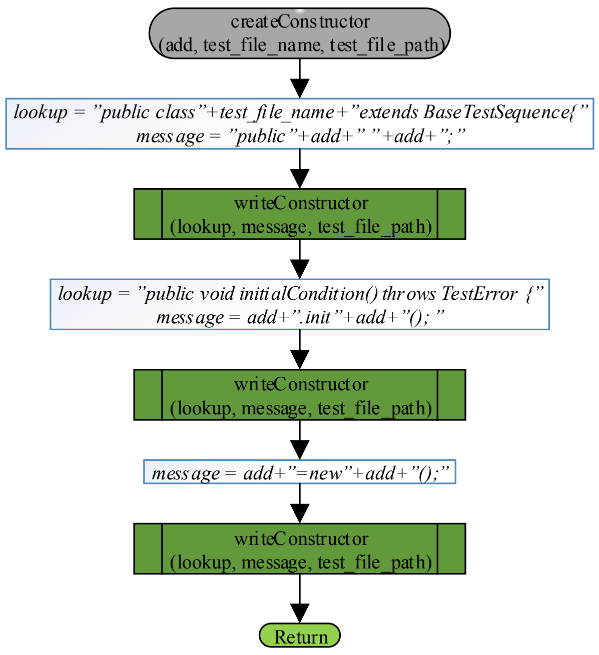

createConstructor
(add, test_file_name, test_file_path)
lookup = "public class"+test_file_name+"extends BaseTestSequence{"
message = "public"+add+" "+add+";"
writeConstructor
(lookup, message, test_file_path)
lookup = "public void initialCondition() throws TestError {"
message = add+".init"+add+"(); "
writeConstructor
(lookup, message, test_file_path)
message = add+"=new"+add+"();"
writeConstructor
(lookup, message, test_file_path)
Return

Figure 33 - createConstructor function

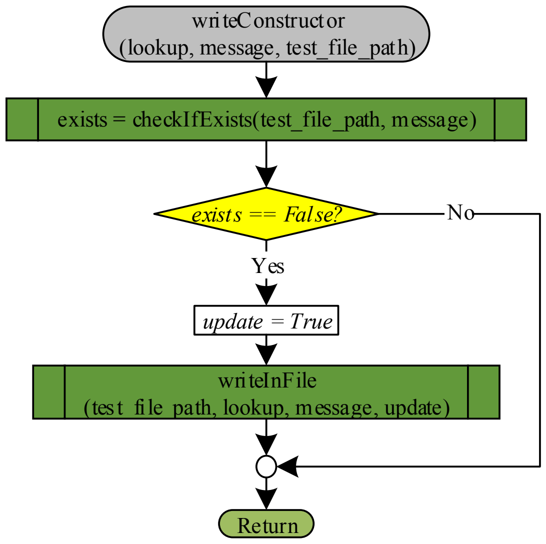

writeConstructor
(lookup, message, test_file_path)
exists = checkIfExists(test_file_path, message)
exists == False?
No
Yes
update = True
writeInFile
(test_file_path, lookup, message, update)
Return

Figure 34 - writeConstructor function

As depicted in Figure 34, the *writeConstructor* function will write information using *writeInFile* function but, before, will check if the same information was already written. The function *checkIfExists* will analyse a file with intention to check if a message exists or not in that file. If the message is already in the file, nothing is done otherwise, the information is written using *writeInFile* function. This process is done in order to obey requirement B.4.

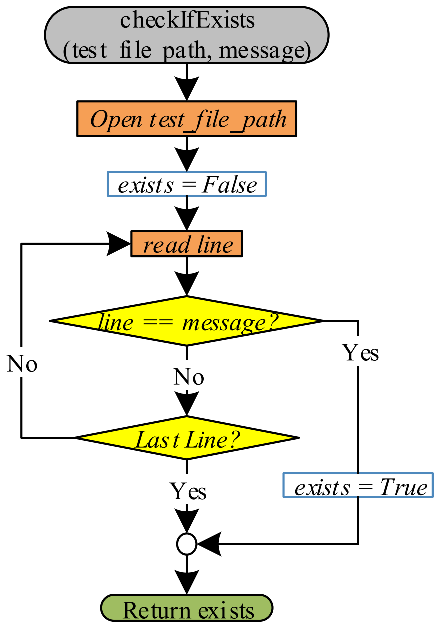
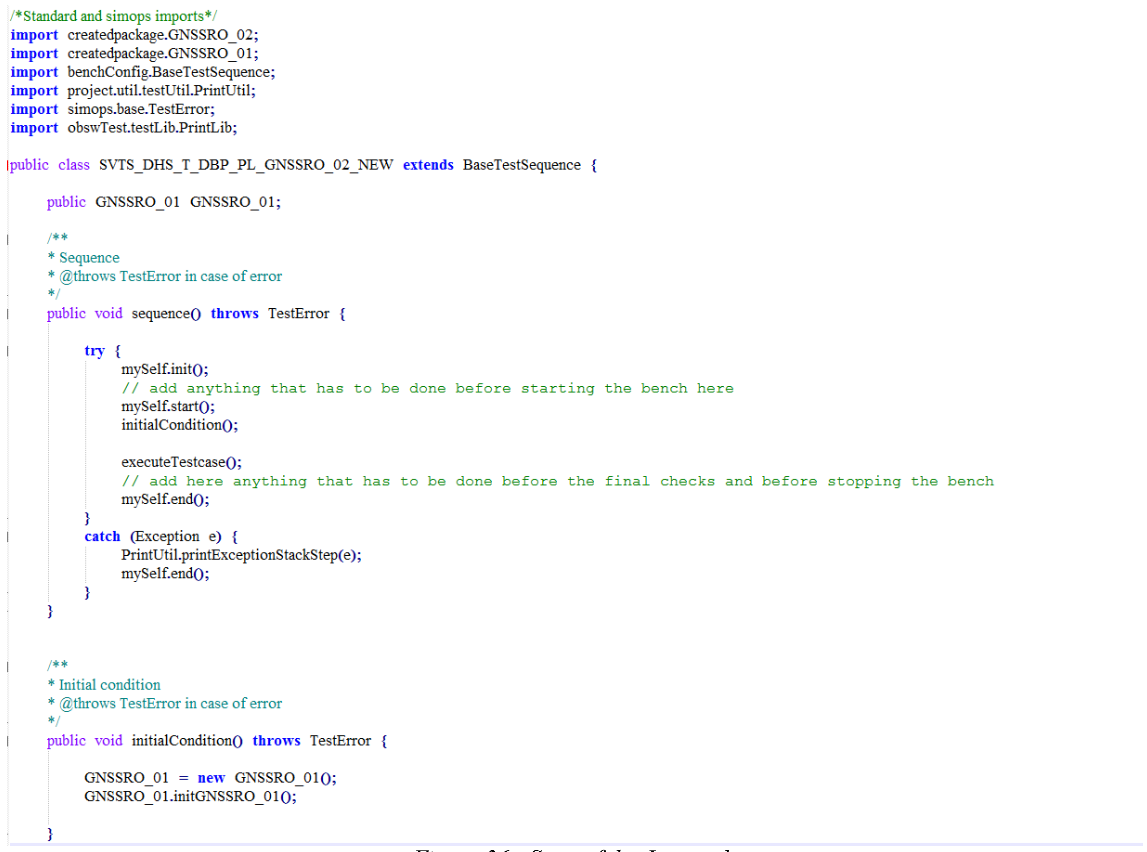


Figure 35- checkIfExists function

Regarding the *checkIfExists* function, as depicted in Figure 35, the function iterates throw all lines in the file and look if a specific message, passed as argument, exists in the file. If the message exists, a Boolean is set to True and returned. Otherwise, is returned False.



```
/*Standard and simops imports*/
import createdpackage.GNSSRO_02;
import createdpackage.GNSSRO_01;
import benchConfig.BaseTestSequence;
import project.util.testUtil.PrintUtil;
import simops.base.TestError;
import obswTest.testLib.PrintLib;

public class SVTS_DHS_T_DBP_PL_GNSSRO_02_NEW extends BaseTestSequence {

    public GNSSRO_01 GNSSRO_01;

    /**
    * Sequence
    * @throws TestError in case of error
    */
    public void sequence() throws TestError {

        try {
            mySelf.init();
            // add anything that has to be done before starting the bench here
            mySelf.start();
            initialCondition();

            executeTestcase();
            // add here anything that has to be done before the final checks and before stopping the bench
            mySelf.end();
        }
        catch (Exception e) {
            PrintUtil.printExceptionStackStep(e);
            mySelf.end();
        }
    }


    /**
    * Initial condition
    * @throws TestError in case of error
    */
    public void initialCondition() throws TestError {

        GNSSRO_01 = new GNSSRO_01();
        GNSSRO_01.initGNSSRO_01();

    }
```

Figure 36 - State of the Java code

When the *createConstructor* function finish its work, we will see that the constructors and objects for the current keyword have been added to "*initialCondition*" method, as depicted in Figure 36. Continuing the filling process of the Java test, the functionality calls the *createStepsList* function.

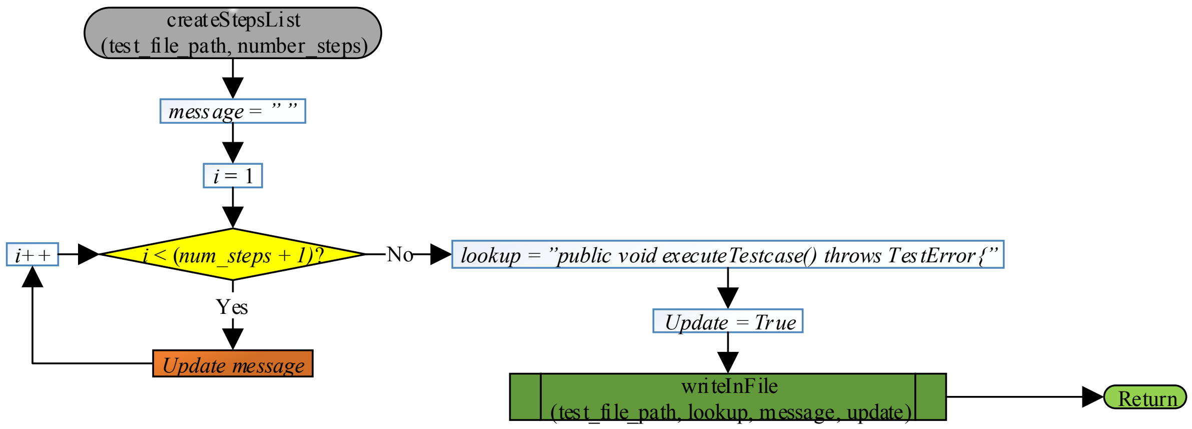
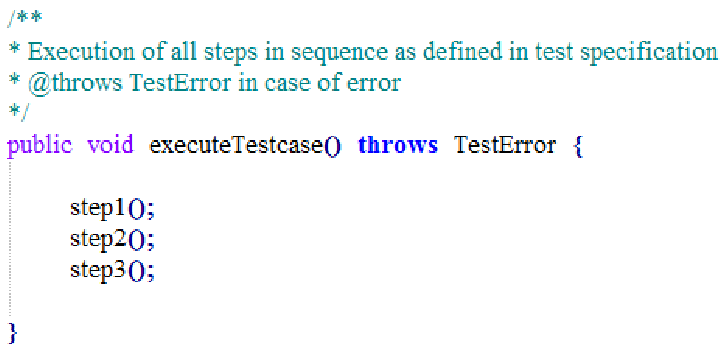

createStepsList
(test_file_path, number_steps)
message = " "
i = 1
i++
i < (num_steps + 1)?
No
Yes
Update message
lookup = "public void executeTestcase() throws TestError{"
Update = True
writeInFile
(test_file_path, lookup, message, update)
Return

Figure 37 - createStepsList function

The *createStepsList,* depicted in Figure 37, functionality is responsible for add the step calls in the "executeTestcase" method.

The *executeTestcase* method drift from the Java template. To provide a suitable and clean image of the code each step will be a Java method. Inside of each method (step) we will have the code related to the keywords that belong to that step.

In order to call these methods, the calls will be made from "*executeTestcase*" method. As depicted in Figure 38, this functionality creates the step calls in this method according the number of steps of the test. As presented before, the number of steps was previously obtained using the *KeywordExists* functionality. Once again, having as example the Table 1, we can see that we have three steps.



```
/**
* Execution of all steps in sequence as defined in test specification
* @throws TestError in case of error
*/
public void executeTestcase() throws TestError {

    step1();
    step2();
    step3();

}
```

Figure 38 - executeTestcase method in Java file

Having the step calls in the "executeTestcase" method, the only thing missing is the function definition of each step.

The *stepKeyword* function, depicted in Figure 39, does two tasks. The first task, is to create the function definition, i.e., create a method that the name have the form *stepx()* being x the current step number. The second task adds the methods calls according the current keyword.

The first task uses again the number of steps. According to the step number, a function definition is created with the same name of the step for example, for step Y: *"public void stepY() throws TestError{ }".*

As already said in previous sections, in order to obtain readable logs and to better understand the output file a code line is added, immediately after the function definition is written, to print the current step in the log file when the test runs: *"PrintLib.printStep("");"*

The second task justifies why the keywords should have the same name as the methods in the Java lib. At this time, we already have the step methods so, the second task iterates throw the keywords in the tabular test and in order to call the method that the keyword contains and since the Java method contains the same name as the keyword we just need to put the keyword inside of the step call and, of course, transforming in a method call. For example, for keyword *switchOnRiu* we will have the method call *"switchOnRiu();".*

Since we have this concerned from beginning related to the keywords names and the methods names in the Java lib, this functionality can build the methods calls relatively easy.

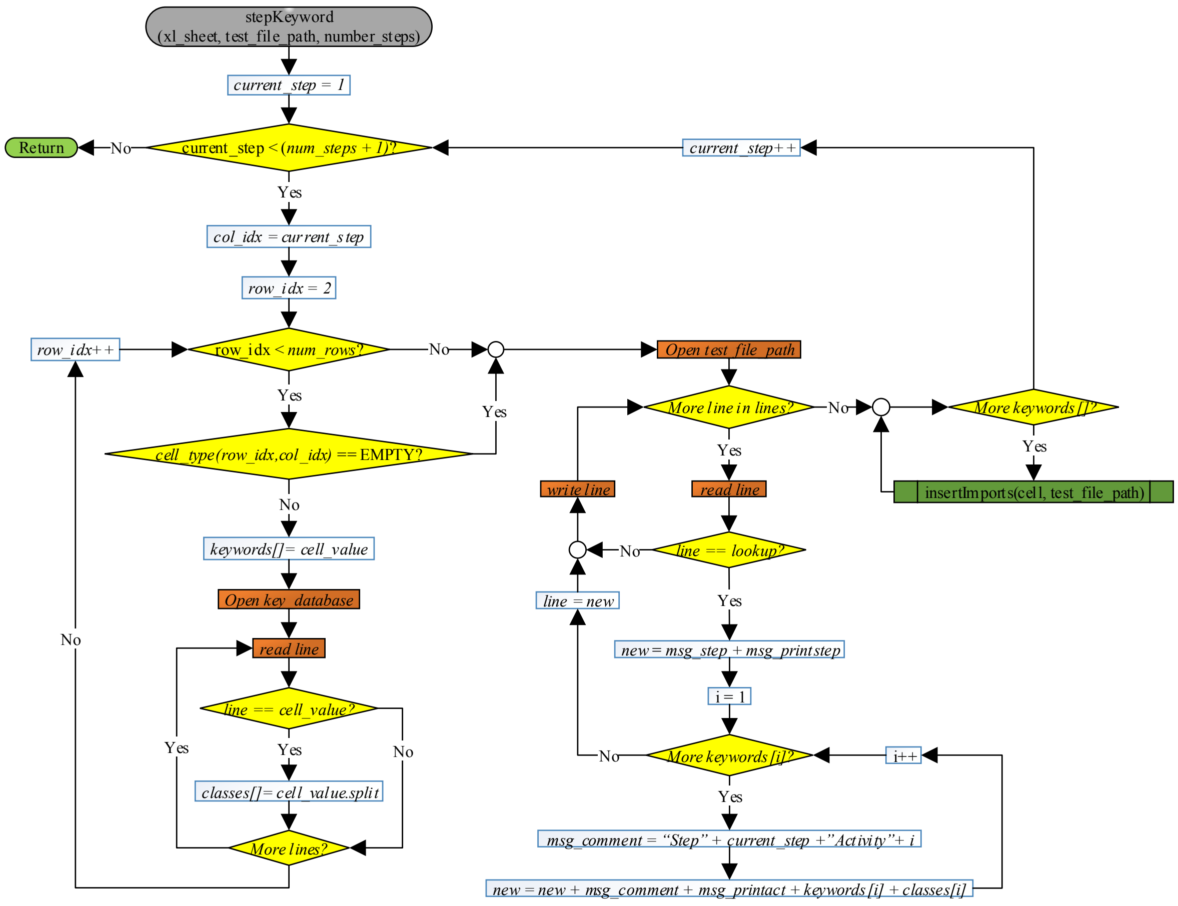

stepKeyword (xl_sheet, test_file_path, number_steps)
current_step = 1
current_step < (num_steps + 1)?
Return
No
Yes
col_idx = current_step
row_idx = 2
row_idx < num_rows?
row_idx++
cell_type(row_idx,col_idx) == EMPTY?
keywords[]= cell_value
Open key_database
read line
line == cell_value?
classes[]= cell_value.split
More lines?
Open test_file_path
More line in lines?
read line
write line
line == lookup?
line = new
new = msg_step + msg_printstep
i = 1
More keywords [i]?
i++
msg_comment = "Step" + current_step + "Activity"+ i
new = new + msg_comment + msg_printact + keywords [i] + classes[i]
current_step++
More keywords []?
insertImports(cell, test_file_path)

Figure 39 - stepKeyword function

Similarly, to *insertConstAndInit* function, the *insertImports* function, in Figure 40, is responsible for retrieving the package of a specific keyword and check if this package was already added to the Java file as an import. In case of the import was already added nothing is done and the sequence ends. Otherwise, the *writeInFile* functionality, is called to perform the "write action" in the file.

The functionality search in keywords database for the current keyword and retrieve the package associated.

To make sure that the current package that will be added doesn't exists in the Java test, the *checkIfExists* function shall be called. If the Boolean returned by *checkIfExists* is False, means that the current package doesn't exists in the Java test. So, the package can be written in the Java test. Otherwise, the functionality continues to the next keyword.

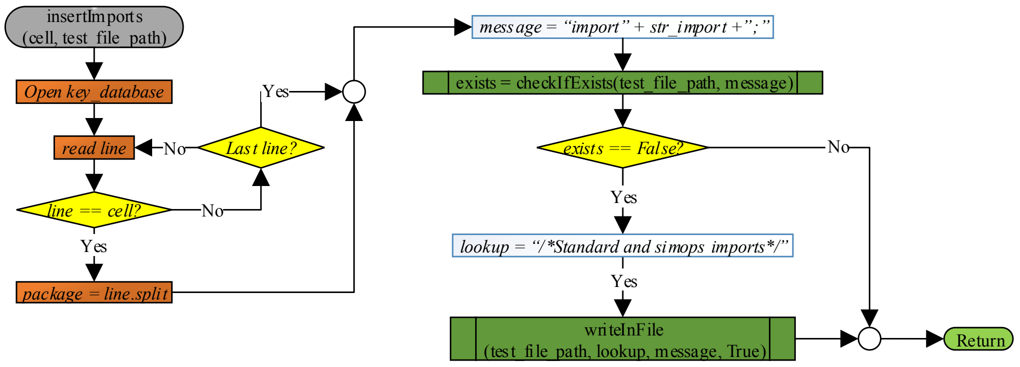

insertImports
(cell, test_file_path)
Open key_database
read line
line == cell?
Yes
No
Last line?
No
Yes
package = line.split
message = "import" + str_import + ";"
exists = checkIfExists(test_file_path, message)
exists == False?
No
Yes
lookup = "/*Standard and simops imports*/"
Yes
writeInFile
(test_file_path, lookup, message, True)
Return

Figure 40 - insertImports function

```
public void step1() throws TestError {

        PrintLib.printStep("");

        /* ---------------- Step 1 - Activity 1---- */
        PrintLib.printActivity("");
        GNSSRO_02.ConfUnitResponses();

        /* ---------------- Step 1 - Activity 2---- */
        PrintLib.printActivity("");
        GNSSRO_02.Modify1553ResponsesSet1();

        /* ---------------- Step 1 - Activity 3---- */
        PrintLib.printActivity("");
        GNSSRO_01.SwitchOnGNSSAndSpy1s();

}

public void step2() throws TestError {

        PrintLib.printStep("");

        /* ---------------- Step 2 - Activity 1---- */
        PrintLib.printActivity("");
        GNSSRO_02.Modify1553ResponsesSet0();

        /* ---------------- Step 2 - Activity 2---- */
        PrintLib.printActivity("");
        GNSSRO_02.Modify1553ResponsesSet1();

}

public void step3() throws TestError {

        PrintLib.printStep("");

        /* ---------------- Step 3 - Activity 1---- */
        PrintLib.printActivity("");
        GNSSRO_01.initGNSSRO();

}
```

Figure 41 - Steps definition

Finally, as depicted in Figure 41, when the K2Java finish we will obtain the Java test ready to run in the SVF.

5 Evaluation and Validation

One important point is that in the description of the dissertation we assert that the component used was the multispectral instrument (MSI) of Sentinel-2 satellite. To provide suitable and understandable information we will test sample features from GNSS-RO component from Jason-CS/Sentinel-6 satellite. This component it's much simpler than MSI and it's enough to provide conclusions about the role of the developed API.

The Jason-CS mission (on the Sentinel-6 spacecraft) is an international partnership between NASA, NOAA, the European Space Agency (ESA), and EUMETSAT. It will continue the high precision ocean altimetry measurements of the Jason-series satellites into the 2020-2030 time-frame using two successive, identical satellites, Sentinel-6A and Sentinel-6B.

A secondary objective of Jason-CS is to collect high resolution vertical profiles of temperature, using the Global Navigation Satellite System Radio-Occultation (GNSS-RO) sounding technique, to assess temperature changes in the troposphere and stratosphere and to support Numerical Weather Prediction.

Figure 42 - Jason-CS/Sentinel6 Satellite

To know more about the GNSS Radio-Occultation sounding technique some information can be found in the attachments [Attachment D].

5.1 *Requirements*

In order to retrieve conclusions of the developed API, some tests will be conducted. The main idea is take a satellite component that was already tested using the manual validation process and produce the same test using the API. With this, we will be able to take conclusions having a sample to compare.

To test a software component, first, we need to know the requirements that the software need to obey. We will provide some requirements taken directly by the user manual [55], [56] and [58]. In a simple way, we will test the Switch-ON and Switch-OFF procedure of GNSS-RO component.

SRS-PLCT-1

The Switch-ON procedure of GNSS-RO component starts sending a Telecommand TC(8,1). This TC is described in the Java lib as *GNSSROSET_SWITCH_ON*.

An event known as *EID_PL_GPSRO_SWITCH_ON_SUCCESS* with event Id(0x3C) should be received to validate that the component switched on with success. This switch-on success event should be received with a 10 seconds timeout.

SRS-PLCT-2

The Switch-OFF procedure of GNSS-RO component starts sending a TC(8,1). This TC is described in the Java lib as *GNSSROSET_SWITCH_OFF*.

An event known as *EID_PL_GPSRO_SWITCH_OFF_SUCCESS* with event Id(0x3D) should be received to validate that the component switched off with success. This Switch-OFF success event should be received with a 10 seconds timeout.

SRS-PLCT-3

Whenever GNSS-RO is disabled, no GNSS-RO DBP 1553 messages are sent to GNSS-RO unit.

SRS-PLCT-4

Whenever GNSS-RO is enabled, GNSS-RO DBP 1553 messages are sent to GNSS-RO unit.

When called, the module shall check the service request bit in the concerned status word for determining Telemetry (TM) data availability.

The bit to check is 7 for set.

If there is no TM available for the concerned RT no further processing will be performed.

Note: SET = '1', RESET = '0'

MSB LSB

							1								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Bit 7: Service Request bit indicates TM data available

Figure 43 - RT Status word

SRS-PLCT-5

Whenever GNSS-RO is enabled, GNSS-RO DBP 1553 messages are sent to GNSS-RO unit.

When there is available TM for the concerned RT the module shall analyse the correspondent Vector Word.

A SET bit in the RT vector word represents the sub-address where a TM data block is available.

If a bit in the received RT VECTOR WORD is SET, the Bus Controller (BC) shall acquire the TM data block from the affected sub-address of the RT.

Note: Each bit in the vector word is defined as a sub-address.

SET = '1', RESET = '0'

MSB LSB

0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Bit 2: TM data block available on subaddress 13
Bit 4: TM data block available on subaddress 11

Figure 44 - RT Vector Word

5.2 *Test Procedure*

According with the requirements presented previously a test procedure has been developed. This procedure has been developed according the test already developed with manual strategy. The manual test, i.e., the pure Java test already developed tests the requirements previously presented. So, we created methods (keywords) that perform these tasks.

Our test will first check for some time the Milbus to check that no messages flows in the GNSS-RO unit. Then, the unit will be switched on and we will see messages flowing. Finally, we switch off the unit and, once again, no messages will flow.

For this test, we develop two steps. The step 1 test the initial values and GNSS-RO Switch-ON procedure. The step 2 test GNSS-RO Switch-OFF procedure.

In order to better understand the activities/expected results and present the information in a proper way two tables will be presented for each step. These tables are used by testers to provide documentation about the test, these tables build the test specification, SVTS.

For step 1, Table 2 resumes all the activities and the expected results for each activity.

	Activities	Expected Results
1	Check the payload Milbus for 1s.	Verify that no GNSS-RO 1553 messages are sent to GNSS-RO unit.
2	Switch-ON GNSS-RO	The event *EID_PL_GPSRO_SWITCH_ON_SUCCESS* is received with a timeout of 10s.
3	Check the payload Milbus for 1s.	Verify that GNSS-RO DBP 1553 messages start after DBP for GNSS-RO is enabled. Verify that TRANSMIT VECTOR WORD message is correctly built, to the correct RT and sent in the correct minor frame.

Table 2 - Activities vs Expected results for step 1

Similarly, for step 2, Table 3 resumes all the activities and the expected results for each activity.

	Activities	Expected Results
1	Switch-OFF GNSS-RO	The event *EID_PL_GPSRO_SWITCH_OFF_SUCCESS* is received with a timeout of 10s.
2	Check the payload Milbus for 1s.	Verify that no GNSS-RO DBP 1553 messages are sent to GNSS-RO.

Table 3 - Activities vs Expected results for step 2

Having the test procedure, we can build the tabular keyword test as depicted in Table 4. At this stage, we assume that the keywords presented complains the Java code that perform the expected check and/or action.

Procedure				
Step / Activity	1	2	3	4
1	SpyPayload1s	SwitchOffGNSS		
2	SwitchOnGNSS	SpyPayload1s		
3	SpyPayload1s			

Table 4 - Tabular test implementation

In our example, each keyword corresponds to an activity, i.e, the keyword *SpyPayload1s* is responsible for spy the payload Milbus for 1s (step 1, activity 1).

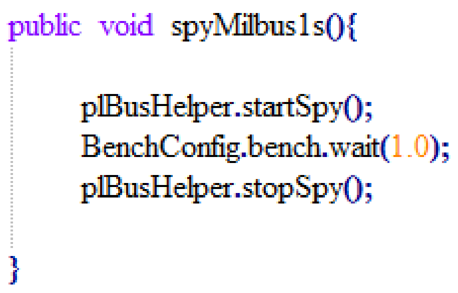
Having the tabular test implemented, we just need to run the application as referred in chapter 4 and retrieve the generated results.

5.3 *Discussion*

In the example presented before, we are assuming that the keywords exist and we just need to run the API but, we need to maintain the idea that, as a standard, the tester needs to follow the following process. When the tester starts to develop a new test shall run the *update_info* method. With this, the tester will have access to the "tester manual" (*Doxygen* page) that present keywords that provide actions and/or checks, at this time the tester can develop the tabular test just filling the table with keywords. Once the test is done, the tester shall run *K2Java* method to translate the tabular test to Java. Of course, that this is the best-case scenario, the activities and checks are developed and each one has a correspondent keyword. In other way, the worst case happens if the tester doesn't find a keyword that provide a check that is present in the test procedure and need first, create a method in Java that perform all the actions and then assign a keyword to this. In this case, the tester need to have fully knowledge on Java and all preambles of the SUT.

The worst case presented can be improved if the SVF team work in line with the keyword-driven approach. This means, if the SVF team knows the concept of keyword can provide methods well defined that can be intended as keywords that can be used by the testers.

One limitation that we verify from this API is the absence of keywords with parameters. Let's take the example of step 1 activity 1: "Spy the payload Milbus for 1s". We want with this, spy the messages that flow in one second as depicted in Figure 45.



```
public void spyMilbus1s(){

        plBusHelper.startSpy();
        BenchConfig.bench.wait(1.0);
        plBusHelper.stopSpy();

}
```

Figure 45 - Code Sample

In terms of keywords we can say that the keyword *spyPayload1s* correspond to the code depicted in Figure 45.

Now, considering that we have a new activity that says to check the payload Milbus for 10s. We can easily understand that a new keyword shall be created with a similar code (changing the parameter of *BenchConfig.bench.wait to 10*). So, we will have two keywords that perform the same action with a slight change. This is not a good programming practice at all.

Continuing the analysis, we can pass to the performance analysis.
We can start the performance analysis comparing the time needed to run a test, comparing, of course, to the "normal" validation process. The time that each test will spend to provide results will be the same because in

both cases we will have actually, Java code running. The difference is that in the standard process we have a Java file with code, functions, etc and in the keyword-translated Java we only have functions calls.

The biggest difference, is in the time to develop a test from scratch. If we want to develop a test using the keyword approach that all activities have a suitable keyword and no more updates are needed in terms of new keywords and Java code this will be much more fast and simpler that develop a pure Java test. In other way, if some keywords need to be developed or updated, more time will be spent, more knowledge about, for example Java, the developer need to possess, etc. The time and even the resources will increase and the gap between use keyword or pure Java will decrease. Finally, if the Java lib contains unsorted methods and all the keywords need to be developed, we will actually be working on pure Java. So, we will have the same work as develop the test in the normal way and then develop tests using keyword. This is not simple and brings more complexity to the validation process.

The last paragraph shows that this API brings more work and costs but, as we already present, the spacecrafts systems are critical systems that reuse code from previous relatives, like Sentinel family.

If this concept is implemented from the beginning of a project or a satellite family the process of validation will decrease in terms of time, costs and number of people in the team.

Another important thing related to performance and more precisely the performance of the API developed is the time to, for example, obtain the manual for testers, i.e., the time spent to run *update_info* method and the time consumed to translate a test, i.e., running *K2Java* method.

To present real data two tests were conducted using a pc with a Intel(R) Core(TM) i7-4710MQ CPU @ 2.50GHz 2.49GHz processor, 16GB of RAM and a 64-bit Operating System.

As we said, our intention was obtain an idea how much time will be consumed to run the two methods that compose our API. First, we analysed the time consumed by *update_info* method to run for different number of classes as shown in Figure 46. These classes contain methods that will be parsed by the method.

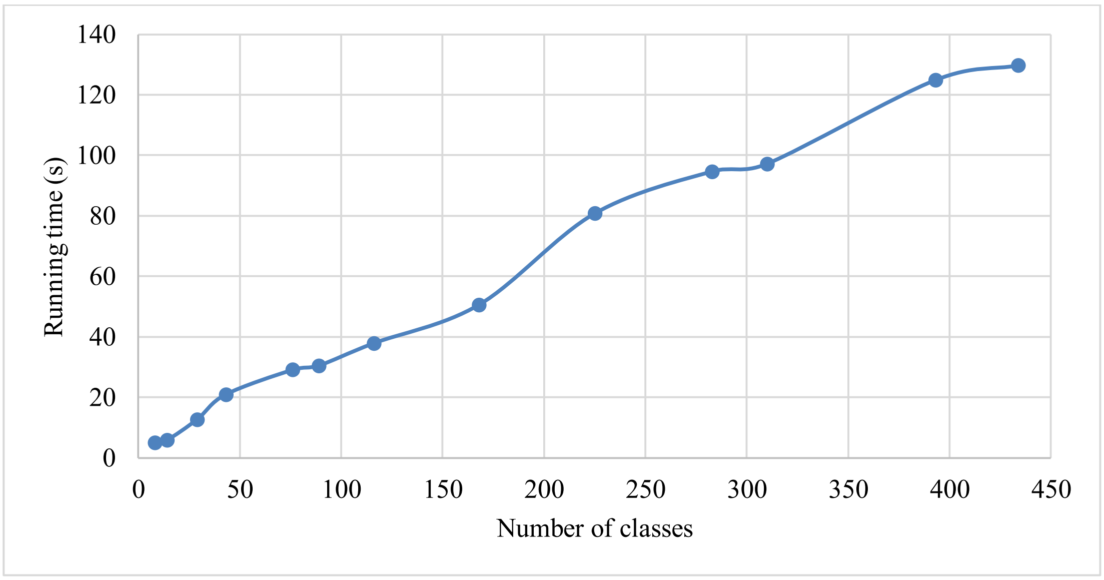


Figure 46 - update_info method running time

As depicted in Figure 46, the *update_info* method spends time in order of seconds. This can be explained because in this method *Doxygen* will run and open, parse and close every class that will be passed. We can see that the graph on Figure 46 presents some discrepancies, this can be explained because each class used for this study doesn't contain the same number of methods.

Regarding the *K2Java* method, we create tabular-keyword tests with different number of keywords. As depicted in Figure 47, the time will be less compared to the previously test. This method is only Python code running, of course this can be improved if we use techniques and concepts to build code to improve performance and time consumption.

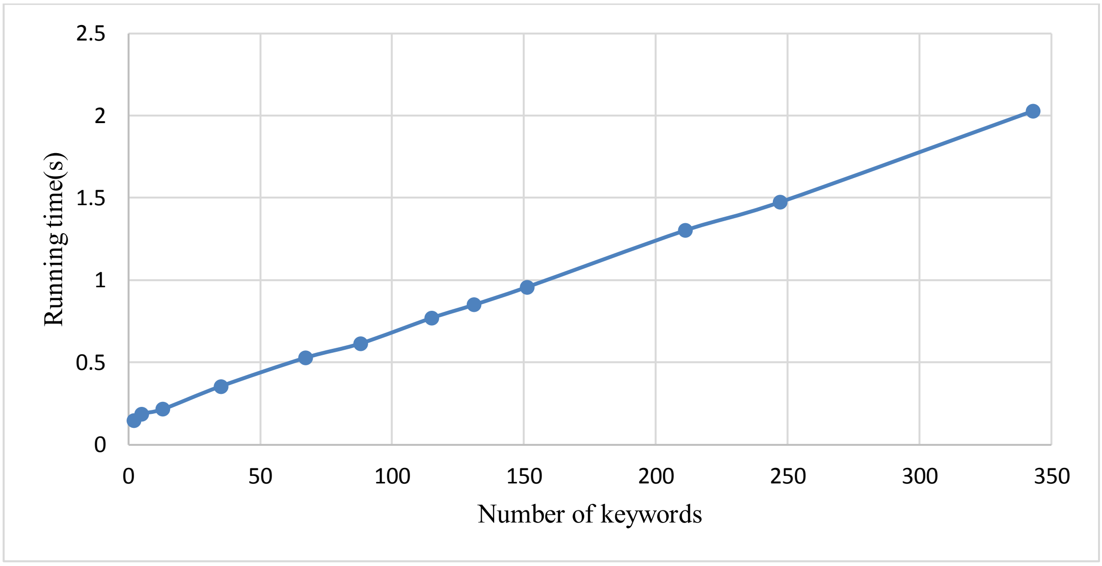


Figure 47 - K2Java method running time

Another conclusion that we can take from the process using the API is that a study shall be conducted to specify the granularity of keywords that the API shall provide. The need of this study is explained by two questions that emerged when looking to the example on Figure 45:

- It's enough to provide a keyword to spy the Milbus with a certain time duration?
- Shall be necessary to provide keywords to start the Spy and stop the spy?

We need to decide the granularity level, i.e., we will have keywords with specific actions or we can agglomerate actions in keywords? This will affect the number of provided keywords and affect the knowledge that the tester must have, for example, it's enough to call a keyword to switch-On a component or, to do the same all the internal actions need to be called? This can be intended as future work.

6 Conclusions

Software touches just about everything in our world today. Today's consumers expect intuitive and reliable technology, and in an increasingly crowded marketplace, small missteps can trigger dissatisfaction abandonment, or even loss of lives. Software testing addresses weaknesses in software development while building scalable development processes to ensure a best-in-class user experience.

This study provides all preambles of Software Testing, reasons about the importance of testing, methods, strategies, manual and automated strategy. Secondly, the development and validation process of satellites was presented to see where our API can be placed to improve the process and automate steps.

Finally, the developed API was presented from scratch, the way how it works, limitations and vantages.

The developed API creates an abstraction layer on top of the Java code. The keywords provide a better way to people with less knowledge on software to read and develop software tests. Of course, that we will need always qualified people to work in the code if a new keyword need to be created to verify new features.

In the presented API, the keywords don't accept parameters. This feature is feasible but regarding the amount of time/resources to do this we can include this in a future work. The number of types of parameters that these SUTs provide is enormous so, the feature shall be well planned.

Regarding the possibly to have keywords with parameters, a functionality shall be developed to test the type of the parameter. This can sound easy to develop but first, a survey of all types should be carried out and the application shall be updated in order not only to test that a keyword exists in the database but also the parameter that it's carrying corresponds to the expected type. In this project, an enormous number of types can be found so, the functionality should be well thought out and shall handle with all types.

From a user perspective, we can see that the way that the available keywords are displayed in the *Doxygen* HTML page can be improved or abandoned since *Doxygen* HTML page have limitations regarding ways of presentation and performance. We can look to the future work as a way to overcome the identified limitations.

THIS PAGE INTENTIONALLY LEFT BLANK

7 References

[1] R. Patton, *Software Testing*. Indianapolis: Publishing, Sams, 2001.

[2] “This Day in History: September 9 | Computer History Museum.” [Online]. Available: http://www.computerhistory.org/tdih/September/9/. [Accessed: 17-Oct-2017].

[3] J. Offutt and P. Amman, *Introduction to Software Testing*, no. 1. Cambridge: Cambridge University Press, 2014.

[4] P. Shukla and D. Mishra, “A composition on software testing,” *Int. J. Tech. Res. Appl.*, vol. 2, no. 2, pp. 27–30, 2014.

[5] G. J. Myers, *The Art of Software Testing*, vol. 1. Canada: John Wiley & Sons, Inc., Hoboken, New Jersey, 2004.

[6] M. Jovanovic, “Software Testing Methods and Techniques,” 2008.

[7] T. Muler and D. Friedenberg, “Certified Tester Foundation Level Syllabus,” 2011.

[8] R. Chauhan and I. Singh, “Latest Research and Development on Software Testing Techniques and Tools,” *Int. J. Curr. Eng. Technol.*, vol. 4, no. Aug, pp. 2368–2372, 2014.

[9] B. Beizer, *Software testing techniques {(2.} ed.)*. Van Nostrand Reinhold, 1990.

[10] S. L. Pfleeger, *Software Engineering: Theory and Practice*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998.

[11] S. Gautam and B. Nagpal, “Descriptive Study of Software Testing & Testing Tools,” *Int. J. Innov. Res. Comput. Commun. Eng.*, vol. 4, no. 6, pp. 2257–2263, 2016.

[12] “Software Testing Levels - International Software Test Institute.” [Online]. Available: http://www.test-institute.org/Software_Testing_Levels.php. [Accessed: 25-Oct-2017].

[13] R. Infotech, “Why testing should start early in the software development life cycle?,” 2015. [Online]. Available: https://www.linkedin.com/pulse/why-testing-should-start-early-software-development-life-infotech.

[14] A. A. Sawant, P. H. Bari, and P. . Chawan, “Software Testing Techniques and Strategies,” *J. Eng. Res. Appl.*, vol. 2, no. 3, pp. 980–986, 2012.

[15] S. H. Trivedi, “Software Testing Techniques,” *Int. J. Adv. Res. Comput. Sci. Softw. Eng.*, vol. 2, no. 10, pp. 433–439, 2012.

[16] K. Mohd. Ehmer, “Different forms of software testing techniques for finding errors,” *Int. J. Comput. Sci. Issues*, vol. 7, no. 3, pp. 11–16, 2010.

[17] U. S. D. of H. S. S. U. S. C. E. R. T. US-CERT, “https://www.us-cert.gov/bsi.” .

[18] W. E. Lewis, *Software Testing and Continuous Quality Improvement*, Second Edi. AUERBACH PUBLICATIONS, 2005.

[19] M. E. Khan and F. Khan, “A Comparative study of White Box, Black Box and Grey Box Testing Techniques,” *Int. J. Adv. Res. Comput. Sci. Appl.*, vol. 3, no. 6, p. 15, 2012.

[20] R. S. Pressman, “Chapter 14 Software Testing Techniques,” *Softw. Eng. A Pract. Approach,* pp. 1–33, 2005.

[21] A. Geraci, F. Katki, L. McMonegal, B. Meyer, and H. Porteous, “IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries,” *IEEE Std 610*. p. 1, 1991.

[22] A. Mailewa, J. Herath, and S. Herath, “A Survey of Effective and Efficient Software Testing,” 2015.

[23] A. Cervantes, “Exploring the use of a test automation framework,” in *2009 IEEE Aerospace conference*, 2009, pp. 1–9.

[24] N. Jenkins, *A Software Testing Primer*. Creative Commons, 2008.

[25] V. N. Maurya and R. Kumar, “Analytical Study on Manual vs Automated Testing Using with Simplistic Cost Model,” *Int. J. Electron. Electr. Eng.*, vol. 2, no. 1, pp. 23–35, 2012.

[26] P. Rathi and V. Mehra, “Analysis of Automation and Manual Testing Using Software Testing Tool,” *Int. J. Innov. Adv. Comput. Sci.*, vol. 4, no. March, pp. 709–713, 2015.

[27] M. Fewster and D. Graham, *Software Test Automation: Effective Use of Test Execution Tools*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999.

[28] S. DESAI and A. SRIVASTAVA, *SOFTWARE TESTING : A Practical Approach:* Phi Learning, 2016.

[29] G. Ghanakota, “Testing frameworks,” in *Introduction to Software Test Automation*, 2012, p. 37.

[30] B. Pettichord, “Seven Steps to Test Automation Success,” *Star*, no. November 1999, 2001.

[31] C. Nagle, “Test Automation Frameworks.” [Online]. Available: http://safsdev.sourceforge.net/DataDrivenTestAutomationFrameworks.htm. [Accessed: 26-Oct-2017].

[32] J. Bach and B. Pettichord, “Lessons Learned in Software Testing: A Context-Driven Approach,” 2002.

[33] A. Divya and D. Mahalakshmi, “An Efficient Framework for Unified Automation Testing : A Case Study on Software Industry,” *Int. J. Adv. Res. Comput. Sci. Technol.*, vol. 2, no. March, pp. 15–19, 2014.

[34] “Guidelines to create a Robust Test Automation Framework,” Philadelphia, 2009.

[35] R. Ramler, S. Biffl, and P. Grünbacher, “Value-Based Management of Software Testing,” in *Value-Based Software Engineering*, S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, and P. Grünbacher, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 225–244.

[36] E. Dustin, J. Rashka, and J. Paul, *Automated Software Testing: Introduction, Management, and Performance: Introduction, Management, and Performance*. Pearson Education, 1999.

[37] M. Grechanik, Q. Xie, and C. Fu, *Software testability: The new verification*. 2009.

[38] P. Tonella, “Evolutionary Testing of Classes,” in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2004, pp. 119–128.

[39] P. Sabev and K. Grigorova, “Manual to Automated Testing: An Effort-Based Approach for Determining the Priority of Software Test Automation,” *World Acad. Sci. Eng. Technol. Int. J. Comput. Electr. Autom. Control Inf. Eng.*, vol. 9, no. 12, pp. 2469–2475, 2015.

[40] M. Kelly, “Choosing a test automation framework,” 2003.

[41] L. Wybouw-cognard, “Test automation framework,” *EP Pat. 1,179,776*, vol. 7, no. 2, pp. 214–219, 2002.

[42] S. Rajeevan and B. Sathiyan, “Comparative Study of Automated Testing Tools: Selenium and Quick Test Professional,” *Int. J. Comput. Sci. Inf. Technol.*, vol. 3, no. 4, pp. 4562–4567, 2012.

[43] H. Kaur and G. Gupta, “Comparative Study of Automated Testing Tools: Selenium, Quick Test Professional and TestComplete,” *J. Eng. Res. Appl.*, vol. 3, no. 5, pp. 1739–1743, 2013.

[44] Y. Kumar, “Comparative Study of Automated Testing Tools: Selenium, SoapUI, HP Unified Functional Testing and Test Complete,” *J. Emerg. Technol. Innov. Res.*, vol. 2, no. 9, pp. 42–48, 2015.

[45] R. N. Khan and S. Gupta, “Comparative Study of Automated Testing Tools: Rational Functional Tester, Quick Test Professional, Silk Test and Loadrunner,” *Int. J. Adv. Techonoly Eng. Sci.*, vol. 3, no. 1, pp. 2348–7550, 2015.

[46] M. Kaur and R. Kumari, “Comparative Study of Automated Testing Tools: TestComplete and QuickTest Pro,” *Int. J. Comput. Appl.*, vol. 24, no. 1, pp. 0975–8887, 2011.

[47] HP, “HP QuickTest Professional software Data sheet,” HP, 2008.

[48] “VectorCAST - Dynamic embedded software testing integrated for SIL, PIL, and Polyspace - MATLAB &amp; Simulink.” [Online]. Available: https://www.mathworks.com/products/connections/product_detail/product_60788.html. [Accessed: 25-Oct-2017].

[49] N. Bhateja, “A Study on Various Software Automation Testing Tools,” *Int. J. Adv. Res. Comput. Sci. Softw. Eng.*, vol. 5, no. 6, pp. 1250–1252, 2015.

[50] G. P. Nokia, “RED Robot Editor User Guide | RED - Robot Editor.” [Online]. Available: http://nokia.github.io/RED/help/. [Accessed: 25-Oct-2017].

[51] J. W. : MSFC, “What Is a Satellite?,” 2015.

[52] “Operating in Space | Satellite Technology | ESOA | Satellite Technology | ESOA.” [Online]. Available: https://www.esoa.net/technology/operating-in-space.asp. [Accessed: 25-Oct-2017].

[53] Z. Sawicka, “Polish Space Instruments,” 2016. [Online]. Available: http://www.cbk.waw.pl/en/images/stories/pdf/2016/space_instruments_1970_2016.pdf. [Accessed: 25-

Oct-2017].

[54] J. Eickhoff, *Simulating Spacecraft Systems*. Stuttgard: Springer, 2009.

[55] Dimitri van Heesch, “Doxygen Manual: Overview.” [Online]. Available: http://www.stack.nl/~dimitri/doxygen/manual/index.html. [Accessed: 25-Oct-2017].

[56] C. Team, “Jason-CS/Sentinel-6 Central Software Requirements Specification,” 2016.

[57] P. Almendra, “Jason-CS/Sentinel-6: Central Software Verification Plan,” Friedrichshafen, 2016.

[58] P. Almendra, “Jason-CS / Sentinel-6 : Central Software Validation User Manual,” Friedrichshafen, 2017.

[59] N. Leveson and C. Turner, “An Investigation of the Therac-25 Accidents,” California, 1993.

[60] R. Beulah and M. Soranamageswari, “Performance and Comparative Study of Functionality Testing Tools : Win Runner and QTP in IT World,” *Int. J. Adv. Res. Comput. Commun. Eng.*, vol. 4, no. 7, pp. 296–301, 2015.

[61] R. Shreshth, “Analysis of Automation Testing Techniques,” 2015.

[62] R. A. Anthes *et al.*, “The COSMIC/FORMOSAT-3 Mission: Early Results,” *Bull. Am. Meteorol. Soc.*, vol. 89, no. 3, pp. 313–333, 2008.

ATTACHMENT A

THERAC-25, A Medical Linear Accelerator, 1986-1987

The Therac-25 was a radiation therapy machine. In layman's terms, it was a "cancer zapper"; a linear accelerator with a human as its target. Using X-rays or a beam of electrons, radiation therapy machines kill cancerous tissue, even deep inside the body.

These room-sized medical devices would always cause some collateral damage to healthy tissue around the tumours. As with chemotherapy, the hope is that the net effect heals the patient more than it harms them.

For six unfortunate patients in 1986 and 1987, the Therac-25 did the unthinkable: it exposed them to massive overdoses of radiation, killing four and leaving two others with lifelong injuries [59].

During the investigation, it was determined that, during the process of calibration, the machine emitted 100 times more energy than the required one.

The case of the Therac-25 has become one of the most well-known killer software bugs in history. Several universities use the case as a cautionary tale of what can go wrong, and how investigations can be lead astray. Much of this is due to the work of Nancy Leveson, a software safety expert who exhaustively researched the incidents and resulting lawsuits. Much of the information published about the Therac is based upon her research paper with Clark Turner entitled "An Investigation of the Therac-25 Accidents". [59]

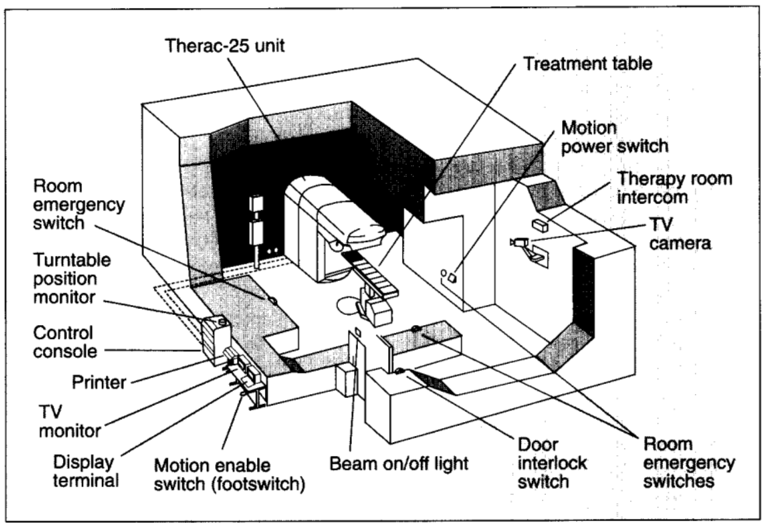


Figure 48 - Typical Therac-25 facility [16]

Patriot Missile Defence System, 1991

The U.S. Patriot missile defence system is a scaled-back version of the Strategic Defence Initiative ("Star Wars") program proposed by President Ronald Reagan. It was first put to use in the Gulf War as a defence for Iraqi Scud missiles. Although there were many news stories touting the success of the system, it did fail to defend against several missiles, including one that killed 28 U.S. soldiers in Dhahran, Saudi Arabia.

Analysis found that a software bug was the problem. A small timing error in the system's clock accumulated to the point that after 14 hours, the tracking system was no longer accurate. In the Dhahran attack, the system had been operating for more than 100 hours. [1]

Intel Pentium Floating-Point Division bug, 1994

Figure 49 - Intel bug

In 1994, Intel introduced the fastest microprocessor of the time, the Pentium. This processor would fail to divide floating-point numbers. For example, by dividing 4195835.0 by 3145727.0 the result presented by the microprocessor was 1.33374 instead of 1.33382, an error of 0.006%. Fortunately, these cases were rare and resulted in wrong answers only for extremely math-intensive, scientific, and engineering calculations.

According with Ron [1] what makes this story notable isn't the bug, but the way Intel handled the situation:

- Their software test engineers had found the problem while performing their own tests before the chip was released. Intel's management decided that the problem wasn't severe enough or likely enough to warrant fixing it, or even publicizing it.
- Once the bug was found, Intel attempted to diminish its perceived severity through press releases and public statements.
- When pressured, Intel offered to replace the faulty chips, but only if a user could prove that he was affected by the bug.

There was a public outcry. Internet newsgroups were jammed with irate customers demanding that Intel fix the problem. News stories painted the company as uncaring and incredulous. In the end, Intel apologized for the way it handled the bug and took a charge of over $400 million to cover the costs of replacing bad chips. Intel now reports known problems on its Web site and carefully monitors customer feedback on Internet newsgroups.

Millennium bug (Y2K), 1999

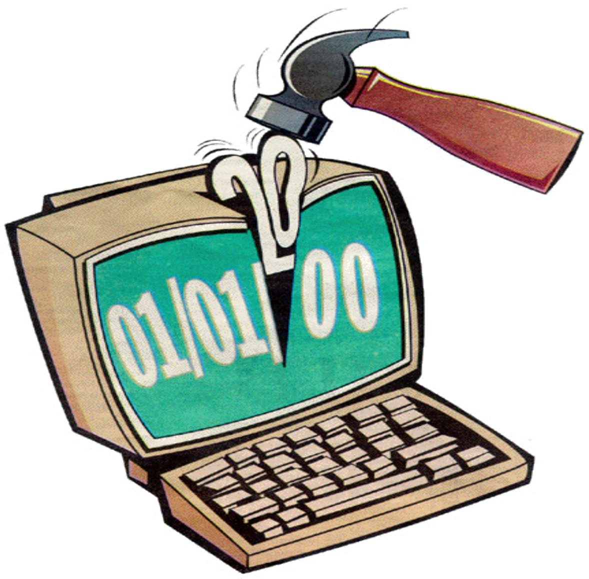


Figure 50 - Millenium bug

The millennium bug was the term used to refer to the predicted problem occurring in all computer systems at the turn of the millennium, 1999 to 2000. Some predicted scenarios of automatically launched nuclear missiles would trigger nuclear war of apocalyptic proportions.

The problem was that all dates were represented by only 2 digits, the programs assumed the "19" in front to form the whole year. Thus, when the calendar changed from 1999 to 2000, the computer would understand that it was in the year 1919 + 00, 1900.

The more modern software, which used to use more current standards, would have no problem dealing with it.

It was found that an enormous number of companies and institutions still had old programs in place, because of the confidence they gained in years of use and their stability.

If the dates were "updated" by 1900, bank customers would come into their applications with negative interest, creditors would become debtors, the risk of aviation accidents would be eminent, etc.

It's estimated that several hundred of billion dollars were spend, worldwide, to replace or update computer programs with this bug, to fix potential Year 2000 failures.

ATTACHMENT B

Methods overview

S.N.	Black Box	Grey Box	White Box
1	The internal workings of an application are not required to be known	Somewhat knowledge of the internal workings are known	Tester has full knowledge of the internal workings of the application
2	Granularity is low	Granularity is medium	Granularity is high
3	Also known as closed box testing, data driven testing and functional testing	Another term for grey box testing is translucent testing as the tester has limited knowledge of the insides of the application	Also known as clear box testing, structural testing or code based testing
4	Performed by end users and also by testers and developers	Performed by end users and also by testers and developers	Normally done by testers and developers
5	Testing is based on external expectations – internal behaviour of the application is unknown	Testing is done on the basis of high level database diagrams and data flow diagrams	Internal workings are fully known and the tester can design test data accordingly
6	This is the least time consuming and exhaustive	Partly time consuming and exhaustive	The most exhaustive and time-consuming type of testing
7	Not suited to algorithm testing	Not suited to algorithm testing	Suited for algorithm testing
8	This can only be done by trial and error method	Data domains and internal boundaries can be tested, if known	Data domains and internal boundaries can be better tested

Figure 51 - Comparison between Three Methods [17]

ATTACHMENT C

Comparison Study

In the literature, is referred that comparison between tools is based in the following parameters (requirements) [46], [43] and [44]:

Recording Efficiency

This parameter is related to the capability of the tool to record information.

The tool can possess recording commands that are inserted in the application to check that works as intended. These commands are called verification points or check points. These are useful to identify whether the website or application functioning correctly or not by comparing a current value for particular property with expected value for the property.

Capability of generation of scripts

This parameter is related to the type of scripts generated by the tool. As an example, QTP only generates VBScripts or Javascripts instead of Selenium (will not be mentioned since is a web testing tool, not relevant for us) that can generate scripts in Java, C#, Ruby, Python, PHP and JavaScript. So, we can observe that Selenium have a high capability to generate scripts than QTP.

Playback of the scripts

When a script is played back, it replays the user actions performed during recording.

Data driven testing

Nowadays data-driven testing becomes a very important part of testing. Instead of recording multiple tests to test multiple sets of input data, it's possible to make the scripts access different sets of input data from external source line data tables, excel sheets, etc. [60]

Several authors consider this framework a relevant parameter when choosing or comparing tools. For us, and since we will apply another framework, don't make sense to evolve this subject. We will consider the capability of the tool to implement the keyword-driven framework since will be the approach used to test our SUT.

Test result reports

After the execution of the test script, it is necessary to get the results of execution to perform effective analysis whether test scripts have passed or failed while running a test suit.

Reusability

Reusing testing logic repeatedly is the ultimate goal of test automation. Automation tools stores the scripts used and can be able to reuse when needed.

Execution speed

As presented before, one of the main reasons to use automated testing and consequently tools is to decrease the time spent on testing.

Makes perfectly sense evaluate the execution speed of each tool in order to choose the fastest one. The reason is simple, less executing time will give results faster and at same point will decrease costs.

Easy to learn

As the name suggests, this requirement is related to the level of expertise that is need to manage the tool and the facility to learn how to use it.

Cost

One of the most important thing to take in account is the price of the tool. As we saw there are two types: Open source and Commercial tools. To the companies, the profit is extremely important so, the price of the tools need to be tacked into account since some tools can be very expensive as we will see.

Knowing the most important parameters, we will present a well-structured table that shows the capability (or not) of each tool to each parameter presented.

Features	IBM Rational Functional Tester	Quick Test Professional (QTP)	Test Complete	VectorCAST	Robot Framework
License cost	Depending of the license can vary between $6800 to $13K	Licensed and very expensive.	~4K for Node-Locked License and ~8K for Floating User License	Depends of the module. For example, VectorCast/Ada license for 3 years ~2K	Free
Application support	Web and Desktop applications.	Client server application only. It also supports add-ons, but user needs to purchase license for them.	Web, Desktop and mobile applications. There are no plug-ins or add-ons to buy.	Several plug-ins and add-ons to buy.	-
Operating System/Platform	Windows and Linux only.	Windows only.	Windows only.	Is designed to support any commercial-quality real-time operating system.	Windows and Linux.
Object Oriented Language support and Scalability	Visual Basic script or JavaScript only.	Supports VBScript or JavaScript.	VBScript, JSScript, DelphiScript, C++Script and C#Script.	C/C++, Delphi, Java, VB, C++.	Python or Java. Other languages supported via a remote interface.
Programming skills Usage	Requires some programming experience.	Quite easy to use. It is quite easy to edit the script, parameterize, navigate, playback and validate the results.	Experience needed.	Experience needed.	Quite easy to use.
Keyword-driven capability	Yes	Yes	Yes	Yes-	Yes

Table 5 -Tool Capabilities [61], [19], [21], [44], [45] and [60]

ATTACHMENT D

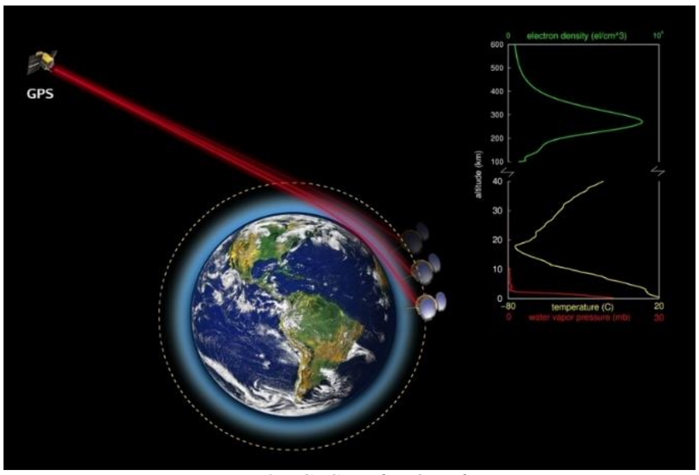


Figure 52 - GPS Radio Occultation

According to [62] the radio occultation (RO) technique, which makes use of radio signals transmitted by the global positioning system (GPS) satellites, has emerged as a powerful and relatively inexpensive approach for sounding the global atmosphere with high precision, accuracy, and vertical resolution in all weather and over both land and ocean. On 15 April 2006, the joint Taiwan - U.S. Constellation Observing System for Meteorology, Ionosphere, and Climate (COSMIC)/Formosa Satellite Mission 3 (COSMIC/FORMOSAT-3, hereafter COSMIC) mission, a constellation of six microsatellites, was launched into a 512-km orbit. After launch the satellites were gradually deployed to their final orbits at 800 km, a process that took about 17 months. During the early weeks of the deployment, the satellites were spaced closely, offering a unique opportunity to verify the high precision of RO measurements. As of March 2013, COSMIC is still providing about 1500 RO soundings per day to support the research and operational communities. COSMIC RO data are of better quality than those from the previous missions and penetrate much closer to the Earth's surface; 70% - 90% of the soundings reach to within 1 km of the surface on a global basis. The data are having a positive impact on operational global weather forecast models.

With the ability to penetrate deep into the lower troposphere using an advanced open-loop tracking technique, the COSMIC RO instruments can observe the structure of the tropical atmospheric boundary layer. The value of RO for climate monitoring and research is demonstrated by the precise and consistent observations between different instruments, platforms, and missions. COSMIC observations are capable of inter-calibrating microwave measurements from the Advanced Microwave Sounding Unit (AMSU) on different satellites. Finally, unique and useful observations of the ionosphere are being obtained using the RO receiver and two other instruments on the COSMIC satellites, the Tiny Ionosphere Photometer (TIP) and the Tri-Band Beacon.