TREBALL FINAL DE GRAU

TÍTOL DEL TFG: Design of a verification and validation framework for an aircraft trajectory computation software suite

TITULACIÓ: Grau en Enginyeria d'Aeronavegació
Grau en Enginyeria de Sistemes de Telecomunicació

AUTOR: Èric Rodríguez Jacas

DIRECTORS: Xavier Prats Menéndez, David de la Torre Sangrà

DATA: July 6th 2023

Títol: Design of a verification and validation framework for an aircraft trajectory computation software suite

Autor: Èric Rodríguez Jacas

Directors: Xavier Prats Menéndez, David de la Torre Sangrà

Data: 6 de juliol de 2023

Resum

Per poder desenvolupar un software de manera òptima, no poden haver-hi errors a l'hora de fer-lo funcionar. Per tant, el procés de verificació i validació que s'aplica és un component crític del seu desenvolupament. És per això que aquest treball consisteix en millorar aquest procés pel cas de Dynamo, un software de simulació de trajectòries d'aeronaus desenvolupat pel grup Icarus de la Universitat Politècnica de Catalunya.

Aquest projecte es marca els següents objectius: comença per conèixer a fons el mètode de testing i la correcció d'errors en software, per tal d'entendre a la perfecció el procés de verificació i validació aplicat als sistemes. Un cop establertes les bases teòriques, és moment per esbrinar tot allò del que disposa el grup Icarus a Dynamo sobre el tema. Per tal de poder millorar el procés que segueixen per corregir errors, primer és important tenir clar què és el que fan actualment i perquè.

Amb tota aquesta informació recollida, el tercer objectiu del projecte és presentar-la als membres del grup Icarus, per tal de que entenguin la importància de tenir un bon procés de verificació i validació. A més, a la pròpia reunió d'exposició del tema, es recullen les possibles peticions que tinguin en quant a millores pel software per tal de que aquestes siguin el més eficients possibles.

Com a objectius finals del treball es proposen les millores a realitzar per tal de perfeccionar el mètode de “testing”, juntament amb un manual amb els passos a realitzar per tal de dur a terme un bon procés de verificació i validació. Així, Dynamo seguirà desenvolupant-se i progressant de la millor manera possible.

Title: Design of a verification and validation framework for an aircraft trajectory computation software suite

Author: Èric Rodríguez Jacas

Directors: Xavier Prats Menéndez, David de la Torre Sangrà

Date: July 6th 2023

Overview

In order to facilitate the optimal development of software, the presence of errors during its operation must be avoided. Consequently, the process of verification and validation plays a critical role in its development. Hence, the purpose of this project is to enhance the procedure mentioned specifically for Dynamo, an aircraft trajectory computation software developed by the Icarus group at the Polytechnic University of Catalonia.

This project delineates the following objectives: firstly, to attain an in-depth understanding of the testing methodology and software error correction, thereby comprehending the verification and validation process employed in systems. Once the theoretical framework is firmly established, the next objective entails exploring the existing resources and practices within the Icarus group concerning Dynamo. To improve the error correction process, it is necessary to ascertain their current procedures and the reasons for carrying them out.

With all this information gathered, the third objective of this project aims to present it to the members of the Icarus group, emphasising the importance of a robust verification and validation process. Moreover, during the topic presentation meeting, any possible requests they may have regarding improvements of the software are collected, aiming to make them as efficient as possible.

As the final objective, this project proposes enhancements to refine the testing methodology, along with a manual detailing the steps needed to carry out a proper verification and validation process.By accomplishing these objectives, Dynamo will continue its development and progress in the most effective manner possible.

Acknowledgments:

To my family, for all their love and support.

To my friends, without whom it would have not been possible to reach the end of this road.

And to David and Xevi, for their patience and help in the achievement of this milestone.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Correction is needed to improve. Everything in life needs corrections to be upgraded and to progress. But these changes need to be in the correct direction and with a sense, or they will be futile. So, what makes these corrections meaningful?

With the evolution and the constant development of software solutions, companies are constantly worried about the perfection of their projects. And this includes correcting and repairing the errors that are found. It is for this reason that the need for quality control when developing software cannot be argued against. Software errors or prolonged developments of some features can lead to frustration of the software developers and less quality in their implementations. That is why companies invest so much of their time and resources in performing a good verification and validation process.

The project aims to create a testing architecture for DYNAMO, an aircraft trajectory computation software created by the group Icarus from UPC[1]. In order to gain a better understanding of the framework, the historical background of the program will be contextualised in Chapter 1. Then, the theoretical framework will be explained in Chapter 2. Initially, the verification and validation process (also known as V&V) will be introduced, defining both concepts and highlighting similarities and differences. Afterwards, the techniques available for the procedure will be described. The amount of existing methods is large, so the focus will be on the ones found to be relevant and useful for Dynamo's situation.

Chapter 3 presents an audit of the current V&V process being carried out in Dynamo. This evaluation includes reviewing the structure of *Fireplace*, the actual testing suite for the software, along with the files available in it. In Chapter 4, the resulting *Fireplace*'s upgrades will be presented. First and foremost, an examination of the available personnel resources will be conducted, along with an analysis of the tasks to which they are designated. Therefore, the possible requests from the Icarus team will be noted down to, then, discuss and assess their feasibility.

With all the members' ideas evaluated and with the knowledge gained from the second chapter, the final *Fireplace*'s upgrades will be proposed in order to improve Dynamo's V&V process. Finally, the conclusions are laid out in addition to the future work required in order to continue expanding and upgrading Dynamo.

CHAPTER 1. CONTEXTUAL FRAMEWORK

In this chapter, a brief explanation of Dynamo will be presented. Also, the situation of testing in the project will be put into context.

1.1. Dynamo

DYNAMO is a suite of 4D trajectory computation software tools, enabling trajectory prediction, optimisation and simulation. It is capable of rapidly computing trajectories using realistic and accurate weather and aircraft performance data. DYNAMO is based on an aircraft point-mass model (3 degrees of freedom) and its design enables it to be used in real-time applications and/or when a large set of trajectories needs to be rapidly generated for simulation or benchmarking purposes.

DYNAMO can be used for dispatching purposes (i.e. computing flight plans) for on-board optimisation computations, for trajectory prediction or simulation. DYNAMO is highly flexible and configurable and allows the user to easily specify a great variety of constraints and objective functions.

An early prototype of this software was initially developed to compute noise optimal trajectories aiming to design advanced noise abatement departure procedures. The core mathematical model and optimisation engine, however, was significantly improved in FASTOP, a Clean Sky project, which aimed to develop an on-board and real-time trajectory optimisation tool for enhanced continuous descent approaches with time of arrival requirements at one or several navigation fixes. In a second Clean Sky project, named CONCORDE, this optimisation engine was embedded on-board into a research FMS and successfully validated through a first campaign using a full-motion flight simulator with qualified crew and a second flight-test campaign using a Cessna Citation II research aircraft.

A series of aviation projects were undertaken to address various aspects of flight management, safety, optimization, and environmental impact reduction. In the Pilot3 project (Clean Sky 2 JU), the primary objective was to generate a software engine for multi-criteria decision support in flight management. This aimed to optimise in-flight aircraft trajectories by considering a comprehensive cost analysis upon arrival. It provided pilots with a range of cost-optimal solutions, particularly when facing unexpected disruptions in the network or flight plan. In the SafeNcy project (Clean Sky 2 JU), the focus was on designing, developing, testing, and validating an on-board function for the Flight Management System (FMS). This function aimed to assist pilots in decision-making during emergency situations characterised by degraded and adverse conditions[1].

More recently the START project (SESAR 2020 JU) aimed to develop, implement, and validate optimization algorithms for robust airline operations, ensuring stable and resilient Air Traffic Management (ATM) performance in disturbed scenarios. CADENZA (SESAR 2020 JU) focused on the development of a detailed broker concept for the European air traffic flow management network, incorporating advanced mechanisms for demand-capacity balancing. The CREATE project (SESAR 2020 JU) aimed to reduce climate and environmental impact through innovative procedures in ATM, introducing alternative trajectories computed in-flight and in real time. Finally, the SIMBAD project (SESAR 2020 JU) addressed the reduction of computational costs associated with performance impact assessment through the application of statistical analysis, machine learning and active-learning meta-modelling within SESAR solutions.

1.2. Current Dynamo testing capabilities

Since the beginnings of the Dynamo group, testing was an important part of the project. The first steps involved a simple bash script, called *Smoke*, in which some arbitrary scenarios of Dynamo were tested. Starting from a group of 4 scenarios, the number of them grew bigger as the project was being developed, ultimately evolving into a much larger bash script called *Forest Fire Test*, in which 96 different Dynamo scenarios were tested. This script also compares the results obtained in a given simulation with previously stored results, usually from the main development branch which is assumed to be the reference solution.

In fact, this is the main validation script currently in use by the software developers for general testing. Months ago, a project was launched so as to become the verification and testing suite for Dynamo3, called *Fireplace*. It is architectured to be a submodule, allowing to run and test many heterogeneous scenarios from Dynamo3. However, despite the existence of *Fireplace*, nowadays the Dynamo team is still using the old Forest Fire Test script.

CHAPTER 2. V&V PROCEDURE

In this chapter, the literature review done about the topic of verification and validation (V&V) will be discussed. These processes will be described, together with the different techniques available to carry them out.

2.1. Software Testing Life Cycle

The Software Development Life Cycle (SDLC) is a series of steps needed in order to perform a good process of software development. Inside this approach, testing is an important part. It should be started as soon as possible and make it part of the development process.

Software Testing Life Cycle (STLC) is an integral part of SDLC, and as well as SDLC does, it defines a sequence of specific activities, in this case, conducted during the testing process to ensure software quality goals are met. STLC starts as soon as the requirements of the program are defined[2], and the steps needed to follow are shown in Figure 1:

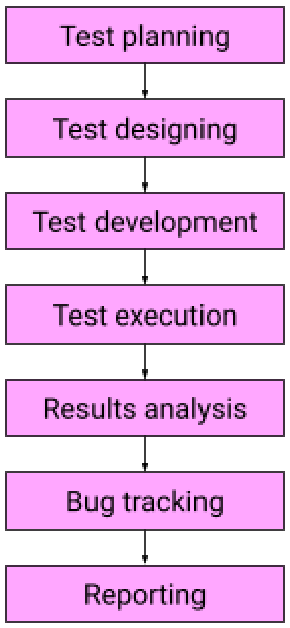

Fig. 1. STLC phases

- Test planning: preparing the testing strategy.
- Test designing: creating the test cases and scenarios.
- Test development: generating the testing environment, trying to replicate the end-user conditions.

- Test execution: running the test cases in the testing environment.

- Results analysis: evaluating the obtained results and reporting the bugs found.

- Bug tracking: analysing the bugs and errors so as to be able to solve them.

- Reporting: it is a post conditional process where reports with the findings are prepared[3].

2.2. Validation and verification

The first issue to consider is that validation and verification are two words not universally agreed upon, as experts differ from a single meaning of both words. In this project, for instance, these words will be very well defined now and it will be the meaning that they will have all across the report.

Verification and validation are independent procedures that, when used together, check that a product, software, system or service meets the specifications and requirements of its intended purpose. That means, to assure that this product, system or service fulfils the user's needs without defects.

2.2.1. Verification

Verification is the process of checking that a software (product, system or service) achieves its goal without bugs. So, it is the procedure of making sure that the developed product is right or not, aiming to detect and aid the designer to find and correct mistakes during the development. It is useful when checking whether the product satisfies its requirements and specifications[4], and it is static testing when talking of software.

Static testing is a software testing technique used to check defects in software applications without the need of executing the code. It involves techniques such as code review, walkthrough or inspection. Verification, in essence, means: are we building the product right?

2.2.2. Validation

Validation is the process of undergoing evaluation of the final product so as to check if it meets the business needs or not. In most of the cases, validation is left until nearly the end of the project when the user meets the system to give her or his final approval. Although, in some development methodologies, the involvement of the user is required throughout the development process. However, if the user's requirements can be precisely defined at the early stages of development, validation could also come in the beginning of the process.

Talking about software, validation is dynamic testing, which includes activities like functional testing, such as unit testing, integration testing, system testing and user acceptance testing. It involves executing the code to validate if the software is high-functional, illustrating this way how the product performs as a whole and its efficiency[5]. Validation, in essence, means: are we building the right product?

In Table 1, the differences between validation and verification are exposed:

VERIFICATION	**VALIDATION**
It includes checking documents, designs, codes and programs.	It includes testing and validating the actual product.
Verification is static testing.	Validation is dynamic testing.
It does not include the execution of the code.	It includes the execution of the code.
Examples include review, walkthrough and inspection.	Examples include all types of testing like smoke, unit, integration, system and user acceptance.
It checks whether the software conforms to specifications or not.	It checks whether the software meets the requirements and expectations of a customer or not.
It comes before validation.	It comes after verification.
It consists of checking documents and files and is performed by a human.	It consists of execution of a program and is performed by computer.

Table 1. Differences between verification and validation

In Figure 2, the different techniques for verification and validation are introduced:

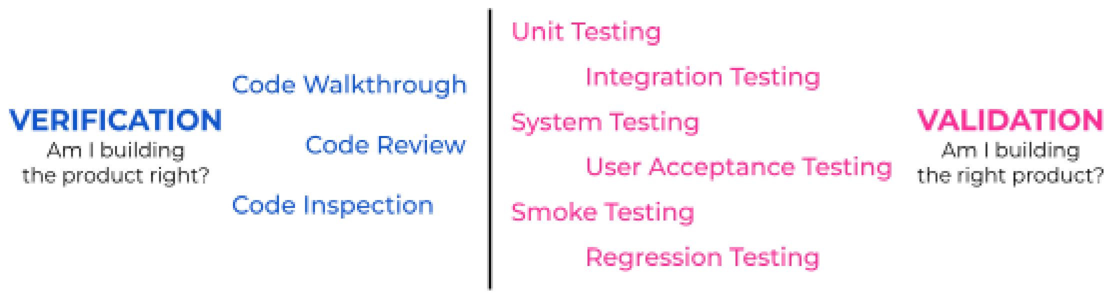

Fig. 2. Verification and validation techniques

2.3. Verification techniques

2.3.1. Code walkthrough

Code walkthrough is an informal code analysis technique and a form of peer review in which a programmer, typically the author of the code, leads a meeting attended by other team members. A copy of the code is given to the team members a couple of days before the assembly, in order to prepare a report with a list of findings. The main purpose of walkthrough is to help team members to learn and gain understanding of the content of the document under review, sharing and discussing their findings with the author, even asking questions if necessary, at the same time they find defects.

Walkthrough enables collaboration and cooperation among members of the team, incorporating multiple perspectives and fostering communication between them. But it also has drawbacks, as there can be areas with defects going unnoticed due to not arising questions. Also, there is a lack of diversity if it is not done properly, as the author leads the meeting and the other participants merely verify if the code matches with what the author is saying, being this situation time-consuming.

2.3.2. Code review

Code review is a systematic examination used, mainly, to find and remove vulnerabilities in the code, such as buffer overflows and memory leaks. Technical experts and peers take part in this well-defined defect detection process, normally without management participation. It is ideally led by a trained moderator, who is not the author of the code. The reviewers have to prepare for the meeting by checking the code and developing a report with a list of findings, and this team gathering can be quite informal or very formal. These technical reviews can have many purposes not only based on finding defects and solving technical problems, but also they are scheduled for decision making or evaluation of alternatives[6].

2.3.3. Code inspection

Code inspection is the most formal type of review, based on rules and checklists, and it is a kind of static testing. The main purpose is to correct the issues in the programming language, finding defects and even spotting any process improvements (if any), so the software can perform at its highest potential. Solving the issues found allows refining the internal structure of the software, as well as security features. A good preparation before the meeting is essential, and it has to be done reading any source documents to ensure consistency.

The process of code inspection entails five steps, being the first one to appoint a team of professionals to inspect and evaluate the code. There are four well defined roles, but there can be other members: the author of the code; the moderator, who leads the code inspection; the reader; and the recorder or transcriber, who compiles the steps of the inspection and makes a report with the findings. Some of the tasks can be done by the same member, but it would be perfect if there are at least two people to have different points of view. For example, the reader and the author can be (and usually is) the author of the code.

The second step is to prepare for the code analysis. The inspector team needs to understand what was meant to accomplish with the software, so the author explains to them the objectives of the project. The inspectors then have to review documents that include metrics of a correct code to be able to compare the language to its ideal formation and catch discrepancies.

The third step is the actual code inspection, where the team follows the stages created by the moderator. The reader goes through the code while the inspectors acknowledge mistakes and the recorder scribes the errors, creating a report to complete every item of the checklist.

Once the meeting finishes, the author has to apply the feedback from the findings to the project, following the instructions that the rest of the team proposed to boost the software's performance. Finally, in the last step, to monitor the success of the evaluation, the moderator hosts a follow-up meeting to check the corrections of the author. Once the code fulfils the moderator's expectations, the verification process is finished.

Code inspection enhances the quality of the product, as corrections can boost the application. Also, reviewing the code aloud can help inspectors (even the author) find defects effectively, saving them time and resources. Moreover, the process can prevent future errors, as engineers can examine how their original configurations lead to defects, allowing them to use the corrected code for future software. But there are also disadvantages, as the process requires extra resources, such as a good group of programmers showing unity with the author.

Despite the technical skills of the team, their productivity and the complexity of the code can lead to needing more time than expected and having to move away some deadlines. In addition, the feedback given by the team members to the author can cause interpersonal conflicts, especially if the results of the report are more extensive than expected for the author[7].

2.4. Validation techniques (software testing)

As mentioned before, the process of verification and validation include both static and dynamic techniques. One of the best known dynamic validation techniques is testing, and it is used not only to help develop automatic test-case generation, but also in the area of fault tolerance and model checking.

When trying to evaluate and verify that a software product or application does what it is supposed to, software testing is the best way. It is a validation process that helps to prevent bugs, reduce development costs and improve performance.

There are many different types of software tests, depending on which specific objectives and strategies are followed. Even a simple application can be subject to a large number and variety of tests. Depending on the available time and resources, a test management plan helps to prioritise which type of testing provides the highest value, as the objective is to find the largest number of defects with the fewest number of tests.

Effective software testing and quality assurance processes can lead to substantial savings, such as time and human resources. When development efforts allocate sufficient resources for testing, it significantly enhances software reliability, resulting in the delivery of high-quality applications with minimal errors[8].

Figure 3 shows the most important validation techniques:

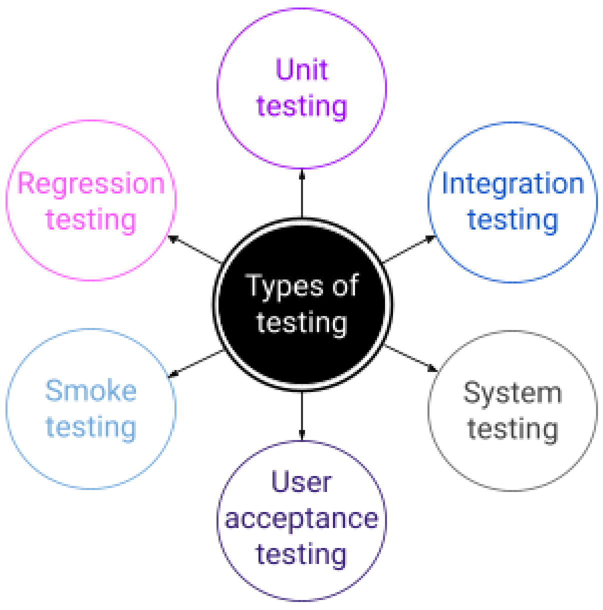

Fig. 3. Types of software testing

2.4.1. Unit testing

Unit testing is the method in which the smallest pieces of code are individually tested against its purpose. In simple words, it means writing a new piece of code (unit test) to verify that the code written (unit) works as intended. This testing methodology is done by the software developers and sometimes quality assurance staff during the development process, and it is done as the first level of testing.

Unit testing is used to design robust software components and to find and fix defects in the early stages of the software development cycle. To do it properly, each test case is tested independently in an isolated environment to assure there are no dependencies in the code. But there has to be some clear criteria, as developers should not make a test for every line of code, since this may take too much time. The focus must be on the lines of code which could affect the behaviour of the software, vitals to the overall performance. This, however, encourages developers to modify the code without concern about how such changes can affect other units functioning.

2.4.1.1. Unit testing methods

Unit testing can be performed in two ways, manual or automated. In manual testing, the tester manually executes test cases by himself. This can be something tedious, especially for tests that are repetitive, but no knowledge of any testing tool is required. On the other hand, there is automated testing, where software testing automation tools are used to automate test cases. With this method, test cases can be saved and repeated as many times as needed without any further human intervention.

For the automation of unit testing, unit testing frameworks are mostly used, as they are helpful in writing unit tests quickly and easily. The majority of programming languages lack inherent support for unit testing through an inbuilt compiler. There is a list of popular unit testing tools for different programming languages, being the most common and important ones:

- Java framework - JUnit
- PHP framework - PHPUnit
- C++ frameworks - UnitTest++ and Google C++
- .NET framework - NUnit
- Python framework - py.test

And there are, in fact, some misconceptions about them:

- It is incorrect to assume that the process of writing code with *Unit test cases* may require additional time, and there may be concerns about time constraints. In reality, this investment of time can result in time savings during the development process in the long term.

- It is incorrect to assume that unit testing will uncover all bugs. The primary purpose of unit testing is to develop resilient software components that exhibit fewer defects in subsequent stages of the Software Development Life Cycle (SDLC).

- It is incorrect to assume that achieving 100% code coverage will mean achieving 100% test coverage. The former does not guarantee the absence of errors in the code.

2.4.1.2. Pros and cons of unit testing

To build a strong castle, one must start with the foundations. By establishing a solid base, the entire structure that follows will be much more stable. That is why, by employing unit testing, we can obtain numerous advantages:

- The process becomes agile: without unit testing, to add new functions or features to the software, the whole code should be tested again. But with it, just the new features should be checked independently.

- Code quality improves: thanks to unit testing, the code becomes robust in design and development.

- Detects bugs early: developers can detect errors (such as flaws, bugs or missing parts) early in the software development life cycle and resolve them.

- Easier changes and simplified integrations: the code is easy for the developer to restructure, make changes and maintain it. It also makes testing the code much easier, as fixing an issue in this stage can fix many other issues that were going to happen in later development and testing processes.

- Reusing code: developers can also reuse code, taking some small parts of it and migrating them to new projects, as they are sure that everything has been checked in advance.

- Lower cost and save development time: the early detection and resolution of bugs during the unit testing phase contribute to notable cost and development time reductions. By identifying and addressing issues at this early stage, developers can alleviate concerns regarding debugging in later stages of the software development process.

But there can be found some drawbacks using unit testing if it is not perfectly executed:

- Hidden bugs: to check that every detail of the software works properly, every line of code should be reviewed. But this does not happen with unit testing, since it will not make the process efficient, causing some bugs to remain hidden.

- Independent tests: unit test only tests sets of data and its functionalities individually, so it will not catch errors in integration.

- Potential time increase: it may be needed more lines of code to test, perhaps, just one line, creating this way a potential time investment.

- More knowledge needed: for a perfect unit testing process, it may have a steep learning curve, as developers will have to learn, for example, how to use specific automated software tools[9].

2.4.2. Integration testing

Integration testing is the process of testing the modules or components of a software when integrated to check that they work as expected. In simple words, this means to integrate or combine the unit tested modules one by one and test the behaviour of them as a combined unit.

Integration testing is normally done after unit testing. Once all the individual units have been created and thoroughly tested, the process proceeds to the integration testing phase. During this stage, the "unit tested" modules are combined to examine their collective behaviour and ascertain whether the implemented requirements are accurately fulfilled. So, this testing process does not happen at the end of the cycle, rather it is conducted simultaneously with the development, as it can be started once the modules are available.

Actually, when applications are developed, they are normally broken down into smaller modules and each developer is assigned to one. Typically, the logic implemented by a developer can differ from another one, so it becomes important to compare the modules among them. But it is also necessary to check the expectations and rendering values in accordance with the prescribed standards.

Furthermore, many times the face or the structure of data changes when integrating modules. Some values can be appended or removed, which cause issues in the later combinations. Moreover, third party tools or APIs (application programming interface) sometimes interact with the software, needing tests to make sure that the data accepted by that API or tool is correct and that the response generated is also as expected.

2.4.3. System testing

System testing is verifying the system as a whole in order to check if it works as expected. This means, to integrate all the modules/components to check that the system meets its specified requirements. System Testing is done after Integration Testing.

System testing should be done in an environment similar to the production one, to assure that the client will be satisfied. It focuses on many aspects, such as external interfaces, complex functionalities, security, performance, operator interaction with the system, usability, load and stress…

To perform a good system testing, requirements and expectations should be clear and the tester needs to know how the application is going to be used and what kind of issues it can face in real-time. In addition, a system test plan should be written and approved, which has to describe what has to be tested along with testing strategies, tools to be used and any other details needed to proceed with the testing[10].

Figure 4 shows the relation between the development of software and the kind of testing needed to be carried out. It is important to highlight that this relationship does not exclude other testing to be done, as smoke testing (see Section 2.4.5.1) or regression testing (see Section 2.4.5.2) can be executed during all the software development life cycle.

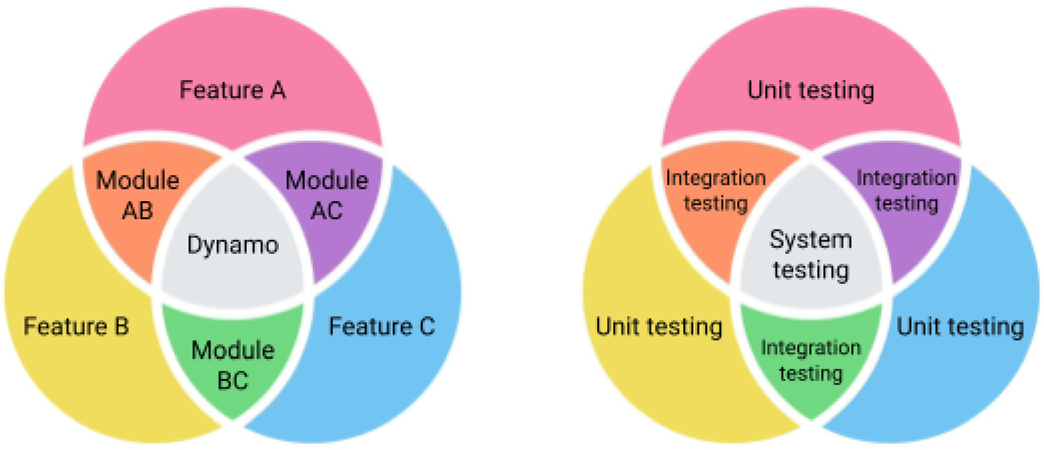

Fig. 4. Relationship between software developed and testing carried out

2.4.4. User acceptance testing

User acceptance testing is the process of testing the software by the user or client to determine whether it can be accepted or not. This is the final testing process, performed once the other ones are done and validated, after system testing normally, as shown in Figure 5.

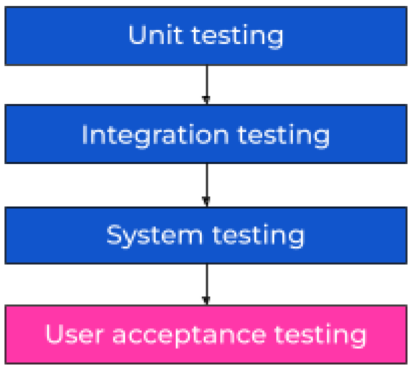

Fig. 5. Main process of testing

The main purpose of this testing is to validate if the clients requirements are fulfilled in the software. This step is usually carried out by the end-users, who are familiar with the business requirements.

Developers and functional testers are technical people who validate the software against the functional specifications, as they interpret the requirements according to their knowledge and then develop the software. But there are some business requirements and processes that are only known by the end-users and they are either missed to communicate or misinterpreted. So, it is very important to carry out a good UAT as the cost of fixing defects after the release is many times greater than fixing it before.

2.4.5. Smoke and regression testing

There are two testing methodologies that stand out among the others, for their simplicity but effective way to help solving defects. Those are smoke and regression testing, employed in almost all the softwares from the very beginning of the project development.

2.4.5.1. Smoke testing

Smoke testing is not an exhaustive testing process, but it is a set of tests that are executed to check if the most important functionalities of a particular build are working as expected or not.

The purpose of this test is not to make an exhaustive process of testing the whole software, but to carry out some validations to the most critical functionalities. The main objective is to make a quick review of the most critical parts to make sure that they work properly. Therefore, to begin, the most critical functionalities need to be identified, in order to make test cases for them.

If in the middle of the smoke test process a defect is found, the following test cases need to be stopped for the error to be solved immediately. This can be done after the system testing, in the user acceptance testing, but also during and before it, such as in integration or even in unit testing, as we can consider it as one of the test cases done during this process.

It is an easy testing process that reduces the risk of defects and tries to find them at an early stage. It also saves money, time and effort, improving at the same time the quality of the system. But this testing is not equal or a substitute for other testing processes, so it can take longer than expected, especially if it is not automated and testers have to execute the cases manually. And, unfortunately, as it does not check all the code line by line, some errors may still be found in a later stage[3].

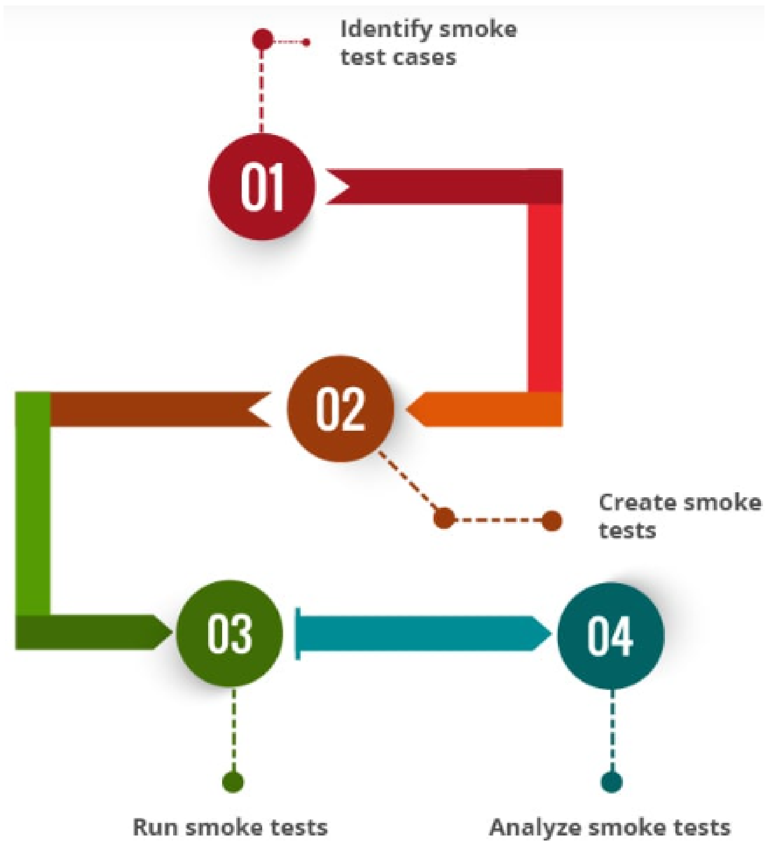

Fig. 6. Process for a good smoke testing approach

The steps needed to be carried out in order to execute a good smoke testing process are the ones shown in Figure 6, and the way to perform them is as follows:

- Identifying smoke test cases: the process of identifying smoke test cases holds significant importance in the execution of smoke tests. It is necessary to determine the minimum number of test cases that cover the critical functionalities of the product so that they can be executed quickly.

- Creation of smoke tests: the smoke tests that have been identified should be used as the foundation for developing corresponding test cases. These test cases are manually created, and if suitable, automation can be employed to create test scripts for their execution.

- Execution of smoke tests: once the smoke tests have been created, they can be executed on the designated build, and the ensuing results can be thoroughly examined.

- Analysis of smoke tests: following the completion of the smoke tests, a comprehensive analysis of the results must be conducted to ascertain whether the build has achieved a successful outcome or encountered a failure[11].

2.4.5.2. Regression testing

Regression testing is a testing process which is applied after a program is modified. It involves testing the modified program with some test cases in order to re-establish our confidence that the program will perform according to the specifications. This means, to ensure that the software works fine with new functionality, bug fixes or any changes to existing features.

To do so, test cases need to be re-executed in order to check if the previous functionalities of the application are still working properly and that new changes have not introduced new bugs. These test cases are normally automated, as they are required to be executed once and again, and doing it manually is time-consuming and tedious[12].

Despite the fact that regression testing seems a simple extension of testing, it is not always the case. There are several major differences between these two processes. The testing process typically begins with a specification, implementation, and a test plan with test cases. During the specification, design, and coding phases, new test cases are added to ensure the program's correct functioning.

Regression testing, on the other hand, starts with a modified specification and program, requiring an update to the existing test plan. All test cases from the previous test plan were previously executed and proved useful in testing the program.

The scope of testing involves checking the correctness of a program and its individual components, as well as their interaction. Regression testing focuses on verifying the correctness of the modified parts of the program, while the unaffected portions do not need to be retested.

Testing time is typically allocated before product development, but regression testing time is often excluded from the tota, product cost and schedule. As a result, regression testing is frequently carried out in a crisis situation, with limited time given to complete the retesting process.

While the testing group usually has access to knowledge about the software development process, including information from the development group, regression testing is often conducted separately and at a different time. Thus, retaining relevant development information becomes crucial for successful regression testing.

The completion time for regression testing should generally be shorter than that of regular testing since only the modified parts of the program need to be tested[13]. Typically, regression testing is applied many times throughout the life of a product, once after every modification is made to an operating product. But there is a workflow very useful to follow when developing a software that helps in doing it in an efficient way. Despite being the possibility of introducing other types of testing in the middle of the process (such as smoke tests), the most common procedure is the one shown in Figure 7:

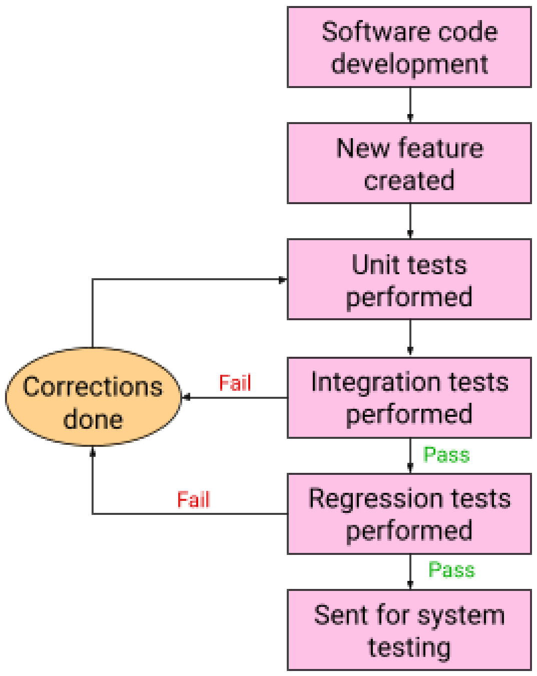

Fig. 7. Steps needed for a good testing process (up until regression)

2.4.6. Automated testing

Automated testing is a process of using software tools to execute tests on software applications. It is an essential part of software development that ensures the quality and reliability of software products. Automated testing has become increasingly popular due to the rapid development and changes in technology and increasing market demands that are leading companies to choose agile methods in software development.

The main objective of applying agile methods is primarily fast adaptation of development processes to changed market demands. Short deadlines in the product supply chain, demand from the companies on one side to shorten the software testing cycle, but on the other side to develop and produce high-quality products. Therefore, companies decide on automating software testing at various levels, which can result in many benefits when introduced efficiently[14].

Automated testing in software development has several benefits, including:

- Saves time and money: automated testing can run tests faster and more frequently than manual testing, which saves time and reduces costs.

- Improves software quality: automated testing can catch defects and errors earlier in the development process, which improves software quality.

- Reduces the risk of errors and defects: automated testing can detect errors and defects that may be missed by manual testing, which reduces the risk of issues in production.

- Increases test coverage: automated testing can test more scenarios and edge cases than manual testing, which increases test coverage.

- Provides faster feedback: automated testing can provide feedback on code changes faster than manual testing, which allows developers to fix issues more quickly.

- Enables continuous integration and delivery: automated testing is a key component of continuous integration and delivery, which allows for faster and more frequent releases.

In summary, carrying out a good process of automated testing can improve the efficiency, effectiveness and quality of software development[15].

CHAPTER 3. FIREPLACE AUDIT

In this chapter, a deeper study of *Fireplace* will be presented. The audit will consist of examining the actual *Fireplace*'s structure, aligning the current testing capabilities with the underlying theory, to ascertain the types of tests currently employed and those that are not fully operational. As commented in chapter 1, *Fireplace* is the verification and testing suite for Dynamo.

3.1. Fireplace structure

Having a first look at the Dynamo repository, there is a folder named *Fireplace* that is in charge of the validation and verification process. So, this is the folder that needs all the attention in order to be able to understand what it is currently doing Dynamo for V&V. The diagram below shows the organisational chart of the folders inside *Fireplace*:

Fireplace
- *cfg*
- *src*
  - ***Fireplace***
    - *Starters*
      - *Embers*
        - *deploy_dynamo*
        - *generate_combinations*
        - *generate_varations*
    - *Tests*
      - *Fire*
        - *additional*
        - *info*
        - *optional*
        - *paths*
        - *summary*
        - *smoke*
        - *fire_main*
        - *scenario*
    - *Utils*
    - *Validate*
      - *Ashes*
        - *validate_binaries*
        - *validate_master_reference*
        - *validate_results*
        - *validate_xml*
      - *Plotters*
        - *DiffPlotterFromFDR*
        - *DiffRatioCalculator*

3.2. Fireplace files

Once the organisation of folders is well defined, let's have a look at the contents of each of them. The focus will be on the python files, as it is there where the validation and verification approach will be defined.

3.2.1. Starters

Embers

- deploy_dynamo: there is, first, the python function *deploy_dynamo* that deploys the dynamo binaries necessary for its run. It takes in the release date of the commit in order to search for the release folder and move it to the directory to store Dynamo's release, together with the maximum verbosity at which dynamo will be deployed and the number of cores to use in the deployment. Also, there is the function *deploy_info*, which gets the deployment/info of the given binary. To do so, it takes in the path to the binary directory along with the name of the binary file, calls the launch script with the specified parameters and gets the output that contains the version info of the binary.
- generate_combinations: script to generate all possible combinations of input parameters for a given set of blocks taking into account the *combinations.csv* file.
- generate_varations: script to generate several combinations files with different variations relying on the *variations.csv* file.

3.2.2. Tests

Fire

- additional: class to save all the additional configuration read from the .ini and its defaults.
- info: class that gets the info of where FIRE is executed (where this class is initiated). This includes the repository name, the repository branch, the commit hash, the commit date and time, the user, the execution time and the machine info. Overall, this class is useful for keeping track of important information about the code execution.
- optional: class to save all the optional configuration read from the *.ini* and its defaults.
- paths: this class organises all paths used in the fire test (base, input, temporary, archive, current, log_file, bin, cfg, output, master_output). The class has methods to create the necessary folders for a new fire test, archive the results of a fire test by moving files from the temporary directory to the definitive output directory, and a method *__update_paths* that updates the directory paths to include the branch, the id, pending changes and other tags using the information. The *create_tree* method cleans or creates folders to prepare for a new fire test. It deletes the current and temporary folders and creates the bin and temporary folders

if they don't already exist. It also tries to remove the log file if it exists. The *archive_results* method moves files from the temporary directory to the output directory and deletes the temporary directory.
- summary: class to print the FIRE results, such as elapsed time, total tests, reference, execution fails, logger fails, result fails, comparison and xml fails.

Smoke

- fire_main: it is the python script that runs the fire tests for Dynamo. The testing process begins with compiling and checking the binaries, for then generating input configs and validating them. If there is no path specified, it will use the default config file. After that, it runs tests and verifies the results, so as to return the summary of the passed tests. Finally, it saves the deployed binaries, inputs and outputs.
- scenario: this python script defines two classes: the first one, Validation, is a simple class just to store validation results. The second one, Scenario, has an execute method to run the binary and store its return code, and a validate method to check whether the output is correct. The *validate* method checks that the binary was executed successfully and that the expected output files exist in the correct directory. It also checks the binary's logger file to make sure it didn't output any errors or warnings. In addition, it has a *validate_config_file* method to check whether the configuration file for the binary is valid or not.

3.2.3. Validate

Ashes

- validate_binaries: takes in the path to search for binaries and a list of strings representing the names of the binaries that must exist. So, the function checks if the required binaries exist in the expected location.
- validate_master_reference: takes as input the path to the parent test folder and a list of scenario IDs. Given this parent test folder (fire folder, smoke folder…), the function searches for the master directory, looks for the latest test performed and checks that the results are available.
- validate_results: contains three functions: the first one is *validate_logger*, which takes a path to a logger file as input and counts the number of fatals, errors and warnings appearing in the dynamo logger. The second function *check_results_exist* takes a path to a scenario directory and a list of output filenames as input and returns a list of booleans indicating whether each of the specified output files exist in the scenario directory. Then, the third function *compare_results_with_reference* compares the result files from both folders, the scenario one and the reference one, to check that all files are equal.
- validate_xml: receives a path to an XML file, searches for the XSD path linked and validates it.

Plotters

- DiffPlotterFromFDR: script that plots two FDR data files, the master and the experimental ones, and compares them. It is mainly used for debugging purposes.
- DiffRatioCalculator: script that revises selected lines of an FDR file to see if they are too different. Moreover, statistics plots are generated, comparing the master/reference archive against experimental/developer archive, in a similar way that the previous one.

3.3. Testing classification

Once all the important python files found in the *Fireplace* folder are identified, a classification should be done in order to know what type of testing is being carried out. According to the previous nomenclature, there are 6 important types of testing that should be done so as to have a good validation and verification process: unit testing, integration testing, system testing, user acceptance testing, smoke testing and regression testing.

From the scripts previously analysed, only the ones regarding the validation and verification process should be mentioned. For example, in the folder "Fire", there are some python functions that work just as auxiliary functions and they do not participate directly in the V&V process, such as *additional*, *paths* or *info*. Taking this into account, the remaining functions are classified in table 2:

FUNCTION	**TYPE OF TESTING**	**REASON**
Validate binaries	Smoke	It is an important check
Validate master reference	Smoke	It is an important check
Validate logger	*None*	It just counts and classifies the loggers
Check results exist	Smoke	It is an important check
Compare results with reference	User acceptance	It can be considered that the reference results are the desire ones
Validate XML	Smoke	It is an important check
Summary	*None*	It only shows the information on screen
Validation	*None*	It just stores the results, but it does nothing with them
Validate	Smoke	It is an important check

Validate config file	Smoke	It is an important check
DiffPlotterFromFDR	*None*	It just plots the results
DiffRatioCalculator	*None*	It just plots the results
Fire main	System	It makes a full coverage

Table 2. Fireplace's functions classification

Having a closer look at Table 2, some files are finally ruled out from the testing classification as they are just tools. They do not participate directly in the verification and validation process, despite the fact that they help in it.

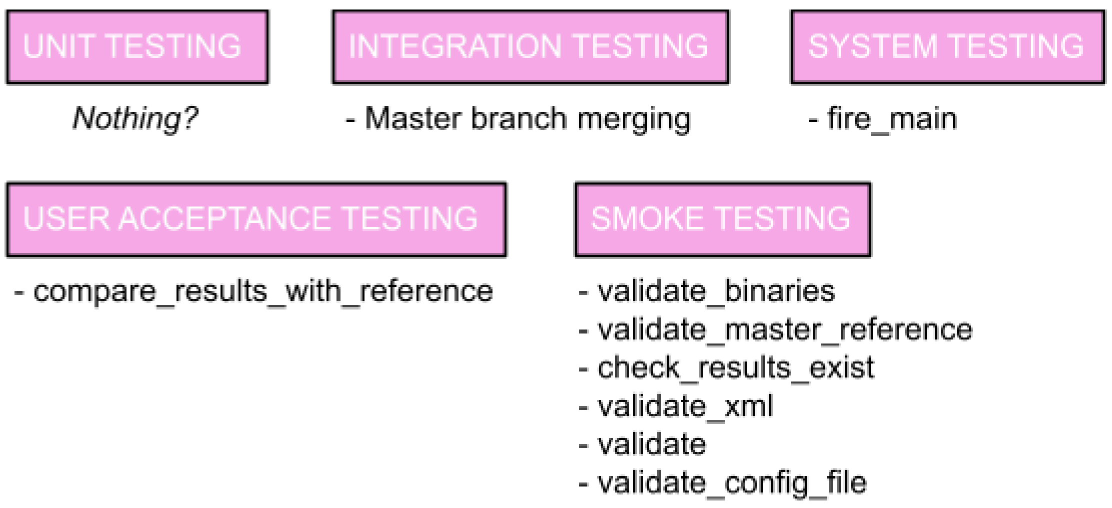

Fig. 8. Fireplace's testing functions classification

Observing the classification scheme of Figure 8, some aspects should catch our attention. Regarding unit testing, there is no trace that it is being carried out on Dynamo at all. The fact that usually unit tests are not saved could be a reason for it. Also, it should be considered that each developer can have their own set of unit tests, as it is something that must not be on the final version of the code but could be useful in future code development.

Despite not finding any python file related to integration testing, there are some tasks that are done in order to have a good integration procedure. When this TFG started, members of ICARUS reported how Dynamo worked, its main functionalities and the validation and verification approach implemented. And this includes the integration of new functionalities. This process is not fully automated and has some steps carried out manually by one member of the team, such as the integration of the new functionality to the master branch and the testing carried out later to check that everything still works as intended.

As this is just a first approach, done by looking at what is inside the *Fireplace* folder, a more accurate assessment should be taken into account in order to have the best possible insight of Dynamo.

3.4. Dynamo team

For establishing a new validation and verification process, or redesigning an existing one, it holds significant importance to gain a comprehensive understanding of the human resources within the team. This requires determining the number of participants involved in the procedure as well as assessing their level of involvement. By obtaining this essential information, it becomes feasible to develop a process that is well-suited to the specific circumstances at hand. In Dynamo, there are three well defined roles:

- Product manager: the product manager plays a crucial role in defining and driving the success of a product. They are responsible for setting the product strategy and vision, based on market research and customer needs. They collaborate with stakeholders to gather requirements and prioritise features. Additionally, they conduct user acceptance testing and monitor product performance. The product manager provides guidance and support to the development team throughout the product life cycle. In Dynamo, there are two product managers.

- Project manager: the project manager is responsible for overseeing the successful execution of projects. He defines project objectives, milestones, and deliverables in collaboration with stakeholders. He creates project schedules, allocates resources, and monitors progress. Risk management and issue resolution are also important responsibilities. The project manager facilitates communication and collaboration among team members and ensures adherence to project management best practices. In Dynamo, there is one project manager.

- Software developers: developers are responsible for writing, testing, and maintaining high-quality code. They work closely with the product and project managers to understand requirements and translate them into technical solutions. They contribute to architectural decisions and collaborate with other team members to integrate components. Developers stay updated with emerging technologies, follow coding standards, and communicate progress and challenges to the project manager. Their expertise ensures the successful development and delivery of software applications. For Dynamo, there are always 2 junior developers with 1 year contract, but the project manager and one of the product managers can also take part in the duties of the developers.

CHAPTER 4. FIREPLACE UPGRADE

In this chapter, the procedure for upgrading *Fireplace* and the validation and verification process will be explained, together with the proposed changes to upgrade the *Fireplace* current capabilities and the new V&V proposed approach.

4.1. Fireplace upgrades

With the study of *Fireplace* done, a session with all Dynamo developers was conducted with the aim to capture the requirements of the new validation and verification process approach. In this session, all the theory regarding V&V was first introduced, in order to initiate them into testing in a more formal way. Also, the results of the V&V audit of Dynamo were presented, giving a special focus to the categorisation of the testing procedures currently in place.

The first idea to clarify was the unit testing. Developers are currently doing it, but not in the most efficient way. Instead of creating unit cases, debugging the code by pieces is the way that they use to check that everything works as expected and to avoid some unit tests. The developers, though, do use unit testing, and it is done as the preliminary part of the integration testing. To make sure that their code does not only work isolated, but also with the whole system, they use some unit tests to check the compatibility. An automation tool is not being used, so this process is fully manual and the tests are not saved. This is the reason why there was no trace of unit testing in the *Fireplace* folder.

Then, talking about the integration testing done, some aspects were deepened in order to understand better what is being done now. First, the code is delivered to the project manager by the developers, once it has been fully tested that it works as intended. Then, the project manager does a code walkthrough to check that the language and code syntax used is appropriate and follows the guidelines. Once finished, they carry out the “merging with the master branch”. This means, to integrate this new functionality to the main development branch of Dynamo. This process is done manually by the project manager thanks to the knowledge he has of Dynamo, but it is a “non comfortable” process as sometimes can be tough to carry out. Then, regression testing is done to assure that none of the previous functionalities of Dynamo have shown new errors. Finally, a system testing is done to check that everything still works as expected.

In the event of an issue arising during the integration testing phase (or during the integration testing process), the responsibility for applying the corrections usually lies with the individual or team primarily responsible for the software's development. However, in Dynamo, it depends on the workload. If the project manager, with a first glance, identifies the error and estimates it to be easily fixable, he will be the one applying the corrections needed. But if at that moment he has tasks to be solved prior to these rectifications, the code will be given back to the original developer to check and resolve the error.

Finally, in relation to the user acceptance testing, there is a greater scope than originally anticipated. The product managers are responsible for this testing. As there is no specific way of doing it by the moment, they use two different options to execute it. One is to do lots of tests in order to check that all of them give possible results; the other one is trying to put Dynamo in tough situations in order to check that the results are the expected. In this part, even though they are also code developers, they act as clients. This means that the process does not include checking if the code is working properly but obtaining results from it, so as to verify that they are correct.

In relation to the mentioned ideas of this section, there are two points to clarify in order to gain a better understanding of Dynamo's V&V methodology and approach. The first one is that this user acceptance testing is differently executed depending on the aim of the code. This means that, as some clients do not know exactly what they want or what should be the expected results, some tolerances should be specified. If the results are clear and well defined, it is obvious that the system should give them in order to accept that it is working properly. But if they are not so well identified, results slightly different from one another can also mean that the system is working satisfactorily. Moreover, in the context of Dynamo, it is given more importance to obtaining the desired results than to prove that the code is perfectly written and has the suitable structure. This means that the team has to be more focused on obtaining the right results but without forgetting about the code structure, that should not be a mess.

4.2. Requirements capture

Once all the ideas were put in order and with the validation and verification process performed in Dynamo clear, there was a discussion in the session in order to capture the requirements of the new V&V approach. But the first aspect to have clear is that, for Dynamo, the amount of code already designed and the staff available in the development of the code should be taken into account. For that reason, perhaps a very strict process of verification and validation is likely not feasible given the human resources of the group. However, a new V&V procedure is definitely required, more adjusted to the existing situation and available resources.

With this in mind, some necessities were discussed:

- <u>Unit testing folder:</u> one reason for not doing unit tests is that they are eliminated just after using it. This makes the developers feel like they are wasting time; therefore, they do not bother creating them. But if there is a specific folder for storing these tests, developers would be more encouraged to make unit test cases. Not all the unit tests should be kept, as this could create lots of problems related to efficiency. But those who are more important, such as test cases for complete functions, could be useful to have for future testing.

- <u>Template for testing:</u> some integration testing or user acceptance testing procedures can be challenging and long to carry out. That is why having a template with some parameters to tune could be a great option to make this process faster.

- Integration testing testbench: not only for the main developer to use, but also for the developers can be a very useful tool. Having a place to make the final tests to their code so as to have it ready for the integration testing can be very important to make the integration easier and to find errors in advance. This testbench could be provided with some useful scripts for testing, at the same time that the users can create new ones and store them there for future purposes.

- Automated system testing: to have a 24 hour control of the system performance, there could be a machine executing massive system testing cases, giving some information about them at the end of each day (such as possible errors or inconsistencies found). This would help in the user acceptance testing work, as the testers will be able to only focus on the most difficult cases.

- Summary tools: also, in relation to the previous mentioned, a tool to simplify and summarise the results of test cases could be useful. At the end of the day, taking as an example the massive system testing, there could be lots and lots of results, most of them useless (these tests are looking for errors; if there are none, the information resulting does not help). So, having a tool capable of simplifying the information and just showing the most important one could save time and resources.

Considering the personnel resources and the way they should approach the V&V process, some explored ideas are not feasible. In the case of STLC, there is no need for carrying out all the steps. The developers now mainly design and implement new features, and those do not require to prepare well in advance a full process of validation and verification, since this can be inefficient.

This situation is also currently prevalent for the unit testing process. The software developers should not primarily focus on a strict process for unit testing, but rather explore the potential of artificial intelligence. Automation tools require expertise and may be more time-consuming in terms of cost. Instead, giving the possibility to AI to generate those unit tests can help in some aspects. This approach not only saves time for developers but also presents an opportunity to learn different and new perspectives on software development through the generated code.

Finally, for the integration testing process, a more automated procedure would be helpful for the code developers as well as for the lead programmer. But it seems not so simple to carry out, as every time a new functionality has to be merged with the master branch, the process is different and unique. The code developed each time employs different binaries, functions and variables that need to be tested in order to guarantee that nothing is generating a new error. In this sense, some tools can be more helpful than just trying to automatise all the process.

4.3. Fireplace proposal

4.3.1. Fireplace new structure

In order to have a well organised *Fireplace*, some aspects should be changed, beginning with the folder structure. Despite not being a complete mess right now, there is a more optimal layout:

Fireplace
- *cfg*
- *src*
  - ***Fireplace***
    - *Unit tests*
    - *Smoke tests*
      - *validate_binaries*
      - *validate_master_reference*
      - *validate_results*
      - *validate_xml*
      - *scenario*
    - *Integration testing bench*
    - *System tests*
      - *fire_main*
    - *User acceptance tests*
      - *compare_results_with_reference*
    - *Tools*
      - *Extra*
        - *additional*
        - *info*
        - *optional*
        - *paths*
        - *summary*
        - *smoke*
      - *Deployment*
        - *deploy_dynamo*
        - *generate_combinations*
        - *generate_variations*
      - *Plotters*
        - *DiffPlotterFromFDR*
        - *DiffRatioCalculator*

This is a preliminary structure based on their actual organisation and taking into consideration their files. All of them are going to be kept, but in a different way. The reasons of this new folder composition are:

- **Unit tests:** this is the folder where all the significant and important unit tests will be saved. The most efficient way to store them is by names and dates, including these into the original functions in order to look for them easily.

- **Smoke tests:** in this folder smoke test files will be saved. There can be subfolders, such as the already created *Validation*, as every smoke test right now is validating. For future and different new smoke tests, new subfolders can be created.

- **Integration testing bench:** here, the most important tests for integration will be saved. Now it is empty as there is not a perfect test for all integration procedures, but it should be filled slowly with the new functions that each developer needs to do when validating their new functions.

- **System tests:** at this moment, there is only a test focused on system testing. But, as the software develops and scales up, there could be more system tests developed that should be stored in this folder.

- **User acceptance tests:** in this folder will be the important files for checking that the software meets the user's requirements.

- **Tools:** finally, the tools that help in the V&V process need to be saved separately. Although they do not participate directly in the approach, they are very important and need to have their own place. There should be, right now, 3 different subfolders, but in the future, depending on the needs of the software, could be more: one for the plotters, which help in visualising the results in a friendly way; one for the functions related to deployment; and finally, another subfolder for the extra functions that do not belong to any of the other folders.

4.3.2. New tests

Not only does the structure of *Fireplace* have to change, but it also needs a high variety of tests so as to check every feature of interest. For this reason, new tests should be added to the folders.

Regarding unit tests, developers should store the tests that they consider the most important or the most useful, in order to be able to use them again easily. They can be useful for checking again some functionalities that could be affected by new functions, furthermore they can be recycled for new functions or parts of the code similar to the previous ones.

For smoke tests, first a deeper explanation is needed. Each test in the folder just needs to be independent of the other ones, not needing to do the first checking steps in every single smoke test. But, for a better working of the system, the smoke tests will do a check of the previous steps, in order to be sure that the smoke test in process is able to be completed.

For integration tests and user acceptance tests, the situation is similar to the unit tests. As of now, there are very few tests stored but, from now on, the software developers (as well as the product managers) should be storing all the important tests used. This will make easier and faster future verifications of the software.

4.3.3. Use cases

Once the development of a new functionality for Dynamo is starting, the developer should ascertain the optimal approach for the validation and verification process, in order to develop it in the most efficient way. Note that the project manager and even the product manager can be involved in this process too. These are the steps that should be carried out when doing the V&V process:

4.3.3.1 Developer unit testing

When creating and implementing a new feature, the code developed has to be checked in order to assure that no errors appear. This should be done with the unit tests stored in the *“Unit tests”* folder of *Fireplace*.

When trying to find tests suitable for the feature being implemented, the developer may realise that there are none. This means that, for the developer’s purpose, no one has found the same situation before. In this case, the developer can make new unit tests in order to properly validate the feature and store them in the folder, helping his colleagues (or even himself) in future software development tests.

Once the new functionality has been finished and the most important lines of code have been tested little by little, a full unit test should be carried out, in order to check that it works properly in an isolated environment. This final unit test can be the most important one and should be stored together with the previous ones.

4.3.3.2. Developer integration testing

To make the process easier for the project manager, some integration tests should be done to make sure that the functionalities that have something in common with the new one have not generated unexpected errors. This can be done either with the integration tests already stored in the folder or with new ones that the developer can create.

In order to be sure that Dynamo works properly before checking any integration test, the starting point for a good integration process is to begin with the most important smoke tests. This process will be useful for the software developers as well as for the project manager. The following manual is recommended to be used in the presented order to ensure a proper flow of function execution and accurate results:

1. Test *validate_binaries*

The *validate_binaries* function is used to check the existence of required binaries in the specified *bin_dir* location. Its purpose is to verify whether all the required binaries exist in the expected location. To use this function, the steps to follow are:

- Provide the *bin_dir* parameter, specifying the path where the binaries are expected to be located.

- Provide the *required_binaries* parameter, specifying a list of names of the binaries that must exist.

The expected function output is a tuple with two values: the success value indicating whether all the required binaries exist and an error message if any error occurs.

2. Test *validate_xml*

The *validate_xml* function is used to validate an XML file against the linked XSD path. The XSD path linked to the XML is searched and the validation process is performed. To use this function, the steps to follow are:

- Provide the *xml_path* parameter, specifying the path to the XML file that needs to be validated.

The expected function output is a tuple with two values: the XML passed validation value indicating whether the XML file passed the validation process, and a message providing additional information or feedback regarding the XML validation.

3. Test *validate_master_reference*

The *validate_master_reference* function is used to verify the availability of expected test results within a parent test folder. The latest test performed is searched within the *master_dir* and the presence of expected results is checked. To use this function, the steps to follow are:

- Provide the *master_dir* parameter, specifying the path to the folder containing all the tests of the same type.
- Provide the *scenarios2test* parameter, specifying a list of IDs representing the expected results.

The expected output is a tuple with three values: the success value indicating if the reference folder exists, the path to the folder containing the expected results, and an error message if any errors occur.

4. Tests for integration process

The integration process mainly consists of checking different scenarios of Dynamo's features so as to assure that the new features developed have not introduced errors. These scenarios are usually available in its corresponding folder, but since normally new features have to be tested, a new scenario specific for this new feature would have to be created. Only for the case that a feature is being corrected (or modified), the scenario needed for the integration process would already be done, stored in the folder.

5. Test *validate_results*

The *compare_results_with_reference* function is used to compare result files from two folders. It identifies files based on the presence of specified *results_tags* in their names and determines whether all files are equal or if there are differences. To use this function, the steps to follow are:

- Provide the *result_tags* parameter, specifying a list of tags to search for when identifying result files.

- Provide the *scenario_dir* parameter, specifying the path to the first folder containing the result files.

- Provide the *reference_dir* parameter, specifying the path to the second folder containing the reference result files.

The expected output is a tuple with two values: True if all result files are equal, False if some files differ, and messages indicating which files differ, if any.

This test can be executed by the software developers as well as by the project manager, always keeping in mind that the results of the tests being executed have to be done previously in order to have them stored in the master branch. This means, if a new feature has been created, there will not be results to compare with, thus the function *compare_results_with_reference* will be of no help. But, on the other hand, if a feature is being corrected or re-edited, the function will be very useful in order to check that it works as expected.

4.3.3.3. Project manager integration testing

Once the developers' testing process has finished, the project manager continues the verification.

To start, the new piece of code has to be revised in order to check that everything has been coded following the rules. To do so, the project manager carries out a code walkthrough, reviewing that the syntax and logic employed has no errors.

Once verified, the main developer runs an integration test, in which the functionality is validated to work properly within the complete Dynamo framework.

With all the main functionalities tested, the project manager needs to do a final system testing to check that the system as a whole, with all the features and dependencies between functions still work properly.

Once all of the above has been validated, the feature is ready to be merged into the master branch.

4.3.3.4. Automated system testing

The system testing process carried out by the project manager is not exhaustive. This means that some errors can still remain, unnoticed. Having someone focusing just on doing tests once and again is inefficient and boring, hence there is a machine working all day long executing system tests. The results are given at the day of each day by the computer, very summarised and focusing mainly on the errors that have appeared (if any). This will help the developers to identify the errors easily so as to solve them quickly. The project manager will be responsible for checking the results periodically and to deliver the errors to be solved to the corresponding software developer (or he will keep it to correct it himself) depending on the workload of the team at that moment.

4.3.3.5. Product manager user acceptance testing

Finally, once the system has been checked that works as intended, the results need also to be tested. Considering the group’s requirements, the results being coherent and making sense can sometimes have a relatively higher importance than the code having no errors. So, making use of specific configs and data, the product managers are in charge of carrying out the user acceptance tests to demonstrate that the system not only has no errors, but the results are also suitable and coherent.

CONCLUSIONS AND FUTURE LINES OF WORK

The importance of having a well structured verification and validation process has proven to be extremely important. The great amount of benefits it provides demonstrates that a V&V process is needed in every software project. It serves as a critical safeguard, ensuring that the developed system meets the intended requirements. Moreover, it helps in identifying and rectifying potential issues in early stages, reducing the required effort and costs, and enhancing the overall quality of the solution.

Contrary to some perceptions that testing is a rigid and predetermined process, it has been demonstrated that it offers a remarkable range of options and adaptable approaches. This flexibility makes testing perfectly adaptable to address specific needs and requirements. And it is this adaptability that has allowed Dynamo to have its own and unique testing process.

The initial idea of the Icarus group of a verification and testing suite for Dynamo was decent enough to fulfil their needs. But generating it requires being quite familiar with the verification and validation process, together with the techniques available for it. As there were no experts on the topic in the team, this made the software developers not use it. But now that it has been demonstrated the importance of this V&V process and with the upgrades made to Dynamo, the developers can truly start benefiting from it.

The focus for the next steps should be getting familiar with the new *Fireplace*, trying to actively use it and taking the most benefit possible from it. As it has been already seen, the need for continuous testing makes us think of this suite as something really important for the future of Dynamo, and the software developers will further optimise *Fireplace*'s functionalities by creating and adding new tests. Furthermore, the team needs to be prepared to scale up the project to meet future requirements, expanding the scope of *Fireplace* as dictated by the evolving project's needs. This proactive approach ensures that *Fireplace* remains fully scalable and capable of adapting to future necessities that the project may face.

As usually happens with software development, Dynamo will never have an ending point: there will always be a new feature or a new implementation available to add to the suite. For this reason, the V&V process also needs to continue evolving, with new future ideas and requirements of the system.

BIBLIOGRAPHY

[1] Dalmau, R., Melgosa, M., Vilardaga, S., Prats, X. “A Fast and Flexible Aircraft Trajectory Predictor and Optimiser for ATM Research Applications.” *International Conference on Research in Air Transportation. ICRAT 2018 - 8th International Conference for Research in Air Transportation*, 26-29, p. 1-8, (2018).

[2] Parihar, Mithelesh and Anu Bharti. “Role Of Software Testing Life Cycle (STLC) In Software Development Life Cycle (SDLC).” *International Journal of Research* 6 (2019): 649-661.

[3] V. Gupta. “Software Testing: Smoke and Sanity”. International Journal of Engineering Research & Technology (2013)

[4] "IEEE Standard for Software Verification and Validation," in IEEE Std 1012-1998 , vol., no., pp.1-80, 20 July 1998, doi: 10.1109/IEEESTD.1998.87820.

[5] J. M. Reddy, S. V. A. V. Prasad. "The role of verification and validation in software testing," 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom), New Delhi, India, 2016, pp. 1298-1301.

[6] A. Alsayed, A. Bilgrami. (2017). “Improving software quality management: testing, review, inspection and walkthrough”. International Journal of Latest Research in Science and Technology. doi: 10.29111/ijlrst.

[7] T. Jaber, M. Abdallah and A. Al-thunibat, "A Proposed Code Inspection Model using Program Slicing Technique", 2020 IEEE 5th International Conference on Computing Communication and Automation (ICCCA), Greater Noida, India, 2020, pp. 275-279, doi: 10.1109/ICCCA49541.2020.9250784.

[8] Yen, J., Banks, D., Black, P., Gallagher, L. J., Hagwood, C. R., Kacker, R. N., & Rosenthal, L. S. (1998, March). “Software testing: Protocol comparison” in Eleventh International Software Quality Week (QW’98) Conference.

[9] P. Hamill, “Unit Test Frameworks,” O’Reilly Media, Sebastopol, 2004

[10] M. Conway, R. Molari. “Guidelines for testing and release procedures”. National Aeronautics and Space Administration (1984)

[11] R. S. G, M. K. H. P and M. A. G. "Smoke Test Execution in Software Application Testing," 2022 Fourth International Conference on Emerging Research in Electronics, Computer Science and Technology (ICERECT), Mandya, India, 2022, pp. 1-7, doi: 10.1109/ICERECT56837.2022.10059686.

[12] W. E. Wong, J. R. Horgan, S. London and H. Agrawal, "A study of effective regression testing in practice", Proceedings The Eighth International

Symposium on Software Reliability Engineering, Albuquerque, NM, USA, 1997, pp. 264-274, doi: 10.1109/ISSRE.1997.630875.

[13] H. K. N. Leung and L. White, "Insights into regression testing (software testing)", Proceedings. Conference on Software Maintenance - 1989, Miami, FL, USA, 1989, pp. 60-69, doi: 10.1109/ICSM.1989.65194.

[14] J. Górski, M. Witkowicz, “Experience with instantiating an automated testing process in the context of incremental and evolutionary software development”. e-Informatica Software Engineering Journal, Volume 5, Issue 1, pp: 51-63, doi: 10.2478/v10233-011-0030-4

[15] Sane, Parth. (2021). “A Brief Survey of Current Software Engineering Practices in Continuous Integration and Automated Accessibility Testing”. 130-134. 10.1109/WiSPNET51692.2021.9419464.