Software Engineering Fundamentals: Software Testing with Jest

Technical Report · November 2024

DOI: 10.13140/RG.2.2.12550.69440

5 authors, including:

Anthony Llena
Saint Louis University
3 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Hans Elijah Viduya
Saint Louis University
2 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Software Engineering Fundamentals
Project No. 1 – Software Testing
Jest

Members:

BRAVO, Matt Danielle U.
BRIONES, Neil Angelo Q.
DEL ROSARIO, Edward Nathan G.
DICTAG, Rouxvick T.
FABE, Milton Junsel C.
LLENA, Anthony R.
VIDUYA, Hans Elijah C.

Class Code: 9374 CS 313
Date of Submission: 25 November 2024

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

1 TESTING CONCEPTS

Jest is a software testing framework designed for JavaScript that offers extensive unit and integration testing tools. Software component testing is vital to ensure quality, as it uncovers defects, faults in codes, and failures in behavior that includes a strategy that defines the procedures to be followed during testing, when they are planned and then carried out, and how much work, time, and resources will be necessary. As part of the testing strategy, software component testing employs several component testing techniques that cover test planning, test-case design, test execution, and the subsequent gathering and assessment of data. Jests' user-friendly configuration from testing supports the whole software testing life cycle, including the design and execution of tests from the tides to test execution, as its philosophy is in line with dynamic testing standards. Jest easily integrates into pipelines for Continuous Integration and Continuous Delivery, supporting automation, regression testing, and defect tracking, encouraging effective test cycles and quick feedback. It offers versatility across several testing dimensions and performs exceptionally well in static, dynamic, white-box, and black-box techniques.

1.1 The Objective of Software Testing

The restricted meaning of "testing" is the period that occurs after development and before deployment. Traditionally, the term "testing" refers to program code testing. Coding, however, is a step that comes after requirements and design, which happen far earlier in the project life cycle. More than testing program code is required because the goal of the computer program project is to minimize and prevent faults.

According to Desikan[1], "The fundamental principles of testing are as follows: (1) testing aims to find defects before the customers find them out. (2) Exhaustive testing is impossible; program testing can only show the presence of defects, never their absence. (3) Testing applies throughout the software life cycle and is not an end-of-cycle activity. (4) Understand the reason behind the test. (5) Test the tests first. (6) Tests develop immunity and have to be revised constantly. (7) Defects occur in convoys or clusters, and testing should focus on these convoys. (8) Testing encompasses defect prevention. (9) Testing is a delicate balance of defect prevention and defect detection. (10) Intelligent and well-planned automation is critical to realizing the benefits of testing. (11) Testing requires talented, committed people who believe in themselves and work in teams."

Now with Jest, the JavaScript Testing Framework offers common unit testing that provides input of a unit of code and matches the expected output [2].

1.2 Software Quality and Project Trade-offs

Although software testing contributes to high-quality goods, it cannot raise a product's quality independently. When testing the software's quality, proper testing interaction should be followed. The process's quality influences the final product's quality; if the other phases' quality is poor and the testing's efficacy is low, the scenario cannot be sustained. The product is likely to be discontinued very soon. As the flaws are discovered closer to the product's release date, attempting to make up for poor quality by focusing more on the testing phase will likely put much strain on everyone.

Similarly, suppose one has a terrible testing phase and naively thinks that other stages are of excellent quality. In that case, one can run the chance of unexpected flaws being discovered at the last minute. Naturally, the best situation is when all stages, including testing, exhibit excellent quality. Consequently, when customers experience the advantages of quality, which fosters improved collaboration and organizational success, the quality of the means aligns with the quality of the result.

The Software features that Jest has to offer is the ability to enable snapshot testing as well as the matched with the saved snapshot and check for matching functionality. Furthermore Jest tests are run parallelly to improve run time [3].

1.3 Defects: Faults and Failures

According to Mantyla et al. [4], "Software project failure is the result of several causes." As the researchers were able to identify 130-185 causes per analyzed failure, they were able to conclude the following: First, their results indicate no single root cause for software project failures. Second, the results consolidate the prior work by showing that a software project failure results from a multidimensional process where people, tasks, methods, and project environment are interconnected. Thus, concluding that lack of cooperation, weak task backlog, and lack of software testing resources where common bridges cause the failures. Third, the bridge causes, and causes related to taks, people and methods were particularly common among the causes perceived as the most feasible target for process improvements. Lastly, the causes of failures and their causal relationships are diverse and depend on the case context.

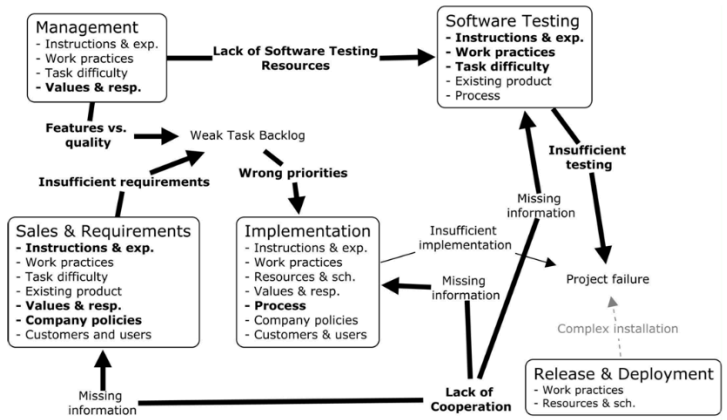


Figure 1. *Common cause and bridging causal relationships found in at least three out of the four cases.*

Data throughout the studies of James Martin, who has explored defect distribution across software development phases said that, "There are errors 56% from the requirements phase, 27% from the design phase, and only 7% from coding errors." This being said that software defects are just a normal part of a software cycle but should be taken with effort.

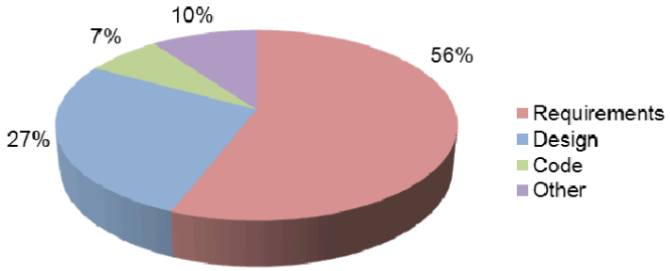


Figure 2. *Distribution of defects in software projects*

Software project failures are common. Even though the reasons for failures have been widely studied, the analysis of their causal relationship still needs to be improved [5]. Because of this, the main objective of software process improvement, which aims to lower development costs, shorten time to market, and enhance product quality, is to eradicate software project failures. Technically speaking, a software project failure is when there is an obvious inability to satisfy the project's goals for scope, cost, time, or quality.

1.4 The cost of fixing a defect

The rule of 1-10-100 started way back in 1992. However, its efficacy still applies until the modern days of software testing. The early stages of fixing a defect during the design phases is the least common cost because it involves updates to documents and plans but not the source code. In this phase, the costs are minimum, that is why the one is applicable such that 1 is a unit of effort or money. Furthermore, if a defect shall be found at the later parts of the development, the cost rises to approximately to the ratio of 1:10 in comparison the early unit of 1 whereas costs are just minimal but if neglected the cost should be multiplied by 10. Lastly, 100, if the defect is found in the final stage of production, whereas the release of the production is too expensive to address, over a hundredfold is the cost to repair it in comparison to the early phase of fixing the defect. The increase is due to the ripple effects of the defects that were mentioned earlier, which are the downtime, customer dissatisfaction, and the company's reputation being in peril, such that legal liabilities are for consideration for such being a failure in this stage [4].

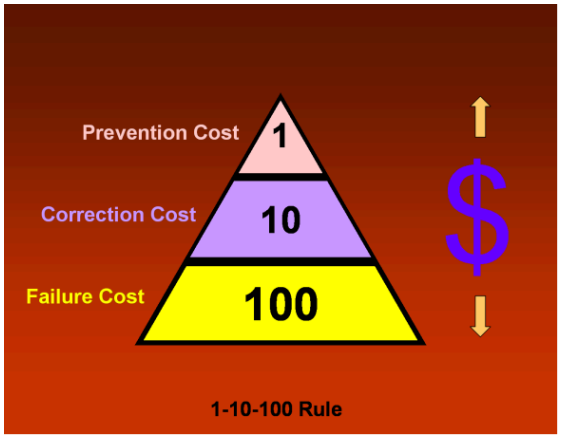


Figure 3. *1:10:100 Rule*

2 TESTING PROCESS

2.1 Software Testing Life Cycle

There are numerous approaches of software testing and when during the software development cycle testing is typically performed. The evolution of software

development is also addressed since it affects the testing life cycle. The following address the several software developments and testing lifecycle: 1) The Waterfall 2) The Agile Methodology. Moreover, the following addresses the testing levels that focus mainly on syntax validation and four levels of testing, namely: 1) Code validation 2) Unit testing 3) Integration testing 4) System testing 5) Acceptance testing.

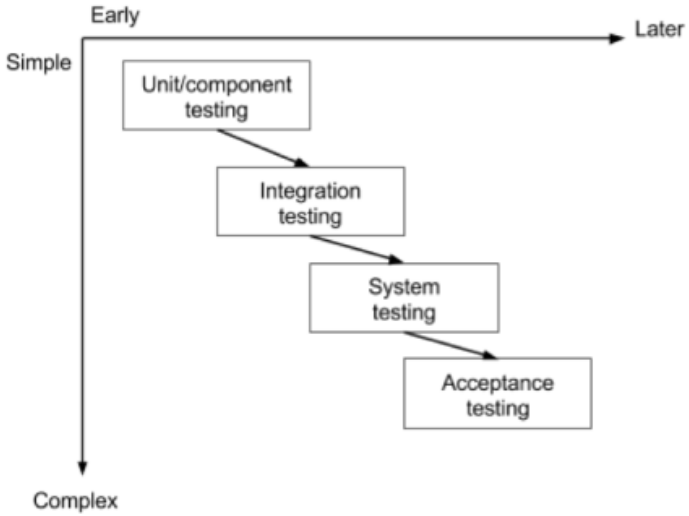


Figure 4. *Testing sequence*

Nevertheless, there are many different phases of development cycles proposed by different authors. The different types of testing that apply to each of the steps offer simplicity that have not shown in the planning phase as a separate entity since it is common for all testing phases." But it is impossible to execute any of these tests until the product is actually built. In other words, the step called "testing" is now broken down into different sub-steps called acceptance testing, system testing, and so on as shown in the figure below, where as different substeps called acceptance testing, system testing, and etc. are still in the case of what all the testing execetuion's relative activities should have done in a matter of only the end of a system development cycle.

Even though the execution of the tests cannot be done till the product is built, the design of tests can be carried out much earlier. Looking at the aspect of skill sets required for designing each type of tests, the people suited to design each of these tests are those who are actually performing the function of creating the corresponding artifact. For example, the best people to articulate what the acceptance tests should are the ones who formulate the overall business requirements. Similarly, the people best equipped to design the integration tests are those who know how the system is broken into subsystems and what the interfaces between the subsystems are-that is, those who perform the high-level design. Again,, the people doing development know the innards of the program code and thus are best equipped to design the unit tests [1].

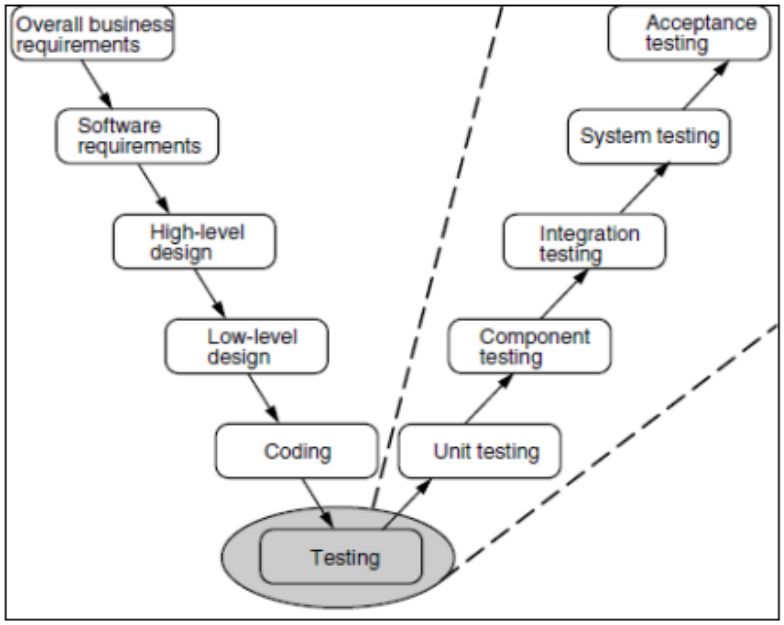


Figure 5. *Phase of testing for different development phases.*

2.2 Defect Injections and Defect Causes

Researchers have recently focused more on developing fault injection techniques deployed in software. Software fault-injection techniques are attractive since they don't require expensive hardware. They can even target complex operating systems and programs using hardware fault injection. If the target is an application, the fault injector is inserted into the application itself or layered between the application and the operating system." If the operating system is the target, the fault injector must be integrated into the system because it is tough to construct a layer between the machine and the system [6].

A system can be examined utilizing a range of high-level abstractions when its dependability is tested at the conceptual and design phases through simulation-based fault injection; implementation specifics are still up for debate. Consequently, the system is simulated using basic assumptions. Determining a system's dependability and the value of fault tolerance solutions is aided by fault injection, which assumes that mistakes or failures follow preset distributions. Also, it provides immediate feedback to the system engineer. Nevertheless, it requires exact input parameters, which are challenging to deliver. The technology, designs and historical data may be difficult to have in scope. Furthermore, it is safe to say that prototype testing allows users to evaluate the system without making conclusions of the system's architecture, producing an accurate outcome. Prototype-based fault injection involves injecting faults into the system to identify dependability bottlenecks, testing how the system reacts to faults, verifying the effectiveness of error detection and recovery mechanisms, and testing the efficiency of fault tolerance mechanisms such as reconfiguration schemes and performance loss. Prototype-based fault injection includes hardware-level injecting of defects (logical or electrical problems) or software-level injectors, including code or data corruption, followed by observation of the results. The system being assessed may be a prototype or a fully operational system. Fault injection is a technique for introducing errors into a system and observing how it responds to those errors. Although it helps to examine the effects of reproduced faults, it has some disadvantages. It is unable to produce accurate reliability measurements, like availability or mean time between failures (MTBF), that are necessary to evaluate a system's performance and dependability in the real

world. Therefore, while fault injection can help understand fault tolerance, it can only partially substitute methods for verifying actual system reliability.

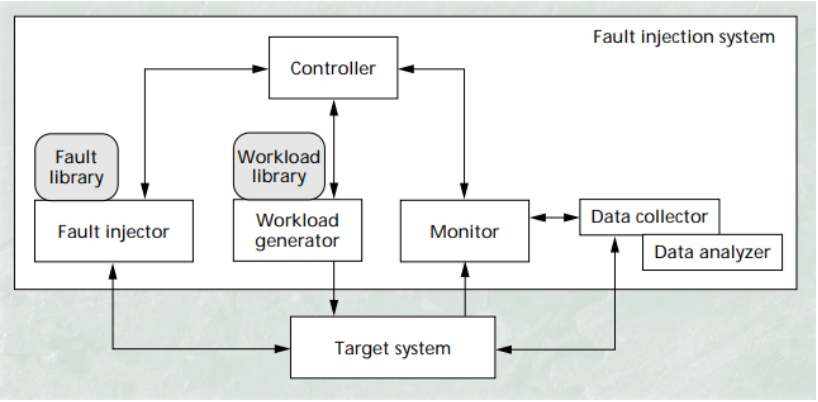


Figure 6. *Basic components of fault injection environment*

Engineers use fault injection to assess fault-tolerant components or systems. Engineers use fault injection to assess fault-tolerant components or systems. Fault injection tests the ability to identify, isolate, reconfigure, and recover from faults. A fault injection environment is shown in Figure 6 and usually consists of a target system, a controller, a monitor, a data collector, a data analyst, a fault injector, a fault library, a fault generator, and a workload library. As the target system carries out workload generator orders (applications, benchmarks, or synthetic workloads), the fault injector introduces problems [6].

The monitor analyzes the implementation of directives and initiates data collection when required. The data collector collects the data online while the analyzer processes and reviews the data offline—the controller experiments. The controller is a physical program that can be used on the target system or another computer. The fault injector may be made with specialized hardware or software. The various types, locations, and times of faults, as well as the required hardware semantics or software structure, are supplied to the fault injector by a fault library. The fault library in Figure 6 provides more flexibility and portability as a stand-alone component.

The sort of flaws you want to produce and the amount of work involved will determine whether you use software or hardware fault injection. Consider the scenario in which you are looking for stuck-at flaws. A hardware injector is preferable in this situation since it allows you to pinpoint the source of the problem. Depending on the problem, it might be expensive or difficult to introduce long-term flaws using software approaches. If there is data corruption, the software technique may be adequate. Both methods can cause errors, such as bit flips in memory cells. Other factors like cost, accuracy, intrusiveness, and reproducibility might affect the approach taken in this case. A list of frequently looked-at errors and injection techniques is shown in Table 1.

2.3 Testing Philosophies

The philosophy of testing is essential to software development, its significance is often emphasized as a business expands from a startup to a larger enterprise. Depending on QA engineers and basic unit tests, it might seem adequate in smaller corporations, but developers must create their own tests in larger organizations. A comprehensive test suite that includes both unit and integration tests ensures that an application's fundamental functionality works as

intended and addresses domain-specific logic. Although Testing may be overlooked for non-production work due to frequent modifications, a robust test suite provides reassurance and lowers the risk of delivering issues in production circumstances. Although Testing is crucial to software development, when a company grows from a startup to a larger organization, its importance is frequently highlighted. While relying on QA professionals and simple unit tests may seem sufficient in smaller enterprises, developers must design their own tests in larger organizations. A thorough test suite that covers domain-specific logic and includes both unit and integration tests guarantees that an application's essential features operate as intended. Rapid changes may cause Testing to be neglected for non-production work, but a strong test suite reduces the chance of delivering problems in production settings and offers reassurance.

2.4 Testing Pipeline

Development teams can modify code with any CI/CD pipeline, and that code is verified by running automated tests before it clears into production. Minimize the ideal CI/CD downtime so that you can release code over and above faster. Continuous integration and continuous delivery — CI/CD for short — is core to modern software development in the fast-paced world of technology today.

CI/CD also produces faster feedback loops with stakeholders so the final product aligns with user expectations. This is a crucial step in any team that wants to build software faster and with better quality. With the CI/CD pipeline, development teams would make changes to code and those changes will go through an automated test first before they are deployed somewhere. Reduce downtime of real CI/CD to release code faster The technology environment is fast-paced making continuous outstanding procedure.

A CI/CD pipeline allows development teams to make changes to code, and the changes are automatically tested before being made available for deployment. Cut down on proper CI/CD downtime to speed up code releases. In today's fast-paced technological environment, continuous integration and continuous delivery, or CI/CD, are more than just business jargon; they are a crucial part of modern software development.

Because it automates all stages of the software development process, from coding to deployment, CI/CD is essential. Teams can provide updates and new features more quickly and frequently because of this automation, which improves the product's responsiveness to consumer requests. Through early error detection, continuous integration and deployment minimize downtime and enhance program quality.

Because it automates every step of a software development process, from coding to deployment, CI/CD is crucial. Teams may release new features and repairs more quickly and frequently thanks to this automation, which enhances the product's responsiveness to user requests. Errors are discovered early on using continuous integration and deployment, which lowers downtime and produces software of higher quality.

To integrate Jest with Azure Pipelines, begin configuring your pipeline to run the tests as part of the build process. The Npm@1 task in the azure-pipelines.yml file can be used to execute the npm run test. However, as Jest runs in watch mode by default, which is blocking for CI, create a separate script in your package.json called test:ci to execute Jest without watch mode with --watchAll=false. This ensures

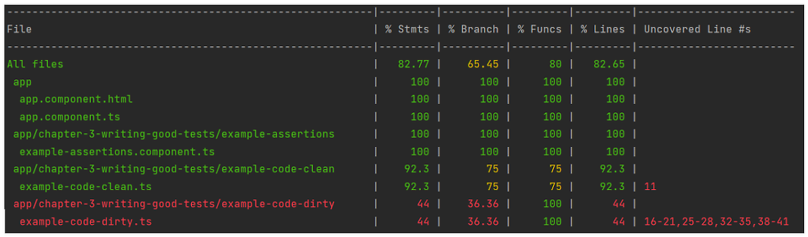
that the tests run as planned and don't loiter in the pipeline. The test results must then be sent by Jest in a manner that Azure Pipelines can comprehend in order for them to be shown in the Azure Pipelines UI. Add the jest-junit reporter to the test:ci script after installing the jest-junit package. To publish test results to a junit.xml file, configure the jest-junit reporter in the package.json. This file will include the test suite results, including information about failed tests, to help with problem diagnosis. Further information about this section is to follow below this paragraph.

3 DEFECT DETECTION APPROACHES

Defect Detection in the context of software testing plays a pivotal role in the success of a software development project. Early defect detection prevents potential system failures that may happen once the system has been distributed to the clients and users. This phase saves a lot of time and money that may be caused by the damages that may occur if this step has been skipped by the development team. It acts as a safety net for development teams when developing software. It prevents potential problems and risk from appearing again and snowballing into an even bigger problem with the software.

3.1 Test Coverage and Testing Dimensions

Software product testing is a very rigorous process in which we must be able to put the program into a series of tests in order to see if there are vulnerabilities and other errors that are not visibly clear for the programmer. It is also an essential part of a testing tool to see what is the metric of the lines of codes that are being tested by a test tool. In jest we can create tests that give us data of how many lines of code that are tested by the testing tool. The coverage test feature in Jest gives you a preview on how much of the code you have been programming is being tested and how much is the percentage of error for each file you have created. The Jest coverage report provides detailed information on your test coverage. To show a coverage report in the console, you can simply use the --coverage flag when running the test. A table containing information about coverage is now shown in the console. It looks like this:



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	82.77	65.45	80	82.65	
app	100	100	100	100	
app.component.html	100	100	100	100	
app.component.ts	100	100	100	100	
app/chapter-3-writing-good-tests/example-assertions	100	100	100	100	
example-assertions.component.ts	100	100	100	100	
app/chapter-3-writing-good-tests/example-code-clean	92.3	75	75	92.3	
example-code-clean.ts	92.3	75	75	92.3	11
app/chapter-3-writing-good-tests/example-code-dirty	44	36.36	100	44	
example-code-dirty.ts	44	36.36	100	44	16-21,25-28,32-35,38-41

Figure 7: *Jest –coverage Display*

There are different types of coverage, Say, for example, every statement has been executed at least once in a 100 per cent statement coverage test, this does not mean that all conditions were checked. In this case, this would be referred to as a branch coverage test. Another type of coverage is line coverage, which specifies the line coverage of all executable source code lines independently of the control flow graph. In the table below, we now see these coverage dimensions along with the accompanying assessment individually and per file.[7]

There are many segments that can be displayed in the coverage section and these "**% Stmts**" which is the percentage of all instructions that were executed at least once by means of tests. [7]. The "**% Branch:**" is where the percentage of all branches whose conditions

were fulfilled at least once by way of tests and thus passed. [7] The "**% Funcs:**" branch represents Percentage of all instructions that were called at least once by means of tests. [7]. While the "**% Lines:**" branch displays the percentage of all source code lines that were run at least once by way of tests. [7]

It may appear a bit unintuitive and this could be fixed with --coverage --coverageDirectory='coverage' where you could have a better visual of the coverage test you have done.

3.2 White-box Versus Black-box Testing Versus Gray-box Testing

Black Box Testing is where the testing is done by a tester where they have no knowledge of the code and will simply do testing for an application for the functionalities of the system if it meets the requirements needed for a certain system without being able to see the code of the system. This is mainly done for test runs of a system to end-users where they will test the system that the programmers are implementing and give detailed and constructive feedback to the programmers about their experiences with the product and what they could do to make it better.[41]

White Box testing is the complete opposite of the Black Box testing in which case where in black box you have no idea of what the code is in the system you are testing it in the case for White-box you have clear and unfiltered information about the system and the implementation of the code. The goal of white-box testing is to see if the code is running properly. From each method such as the loop parameters conditional loops switch cases and such are being tested here and it needs to run properly so that it may be able to have a proper assertion if the system is ready to be rolled out for beta testing.

Gray-box is a bit of both world of black box and white box testing you have knowledge of the code but not entirely the whole code the job of the tester here is to test the application what are the API's being utilized and such[41]

In the context of Jest there are ways in which you can employ the testing types. White-box testing we can use jest.fn or jest.spyOn to inspect and isolate a specific logic to identify any potential risk and errors of that certain area, Also we can use the coverage tool of jest to check what are the percentages of the unchecked statements and logics of the code to be able to test those areas manually through human intervention. For gray-box testing we can use hooks such as beforeAll or afterAll and inside these scenarios we can write tests which will reflect real world scenarios in which the code may be put into a stress test and test the interaction of the modules with partial knowledge of the code.For black-box testing we can use Supertest tool for API Testing and Puppeteer for UI Testing and we can create end to end test with no knowledge of the code. We can also use Boundary Value Testing or Equivalence Partitioning as a way of testing the system to find any particular errors with input values and what errors may occur when the input values are in the extremities such as the maximum and minimum value

3.3 Scripted Versus Non-scripted Tests

Scripted Tests in Jest are where the tester uses controlled variables where the scenarios of the test code is done over and over again with the goal of verification of the functionality, its goal is to see whether through various repeated tests they can check whether the

system has deviated from the expected output of the code to another output. It is focused on a targeted part of a code which is then put into a repeated and automated test with the goal of checking whether the system returns different output from the expected result.[15]

There are ways to implement scripted tests in jest one of which is the test blocks or it blocks. 'It' blocks are used for an individual case test definition. Alternatively you can use the 'test' as an alternative to an it block. If we are to combine the above two, we use 'describe' with a callback for creating a new test module, and then we use 'it' inside that describe block to create one or more individual test.[8]

```
describe(Jest()', () => {
it('should be the best framework', () => {
expect(Jest.bestFramework.toBeTruthy());
});
});
```

Figure 8: *'it' Test Block*

Additionally you can use describe blocks to organize jest test cases in logical groups. The 'describe' blocks help us create certain divisions in the test suite called blocks.. These blocks can be helpful in many cases when we have a group of test cases that need to be put together since they use the same method.[19]

```
describe("Filter function", () => {
// test stuff
});
```

Figure 9: *Describe Test Block*

Scripted test cases are best used when we are dealing with code blocks or groups that can be tested accordingly and where each test is named descriptively to specify its purpose and ultimately automate the testing of these code blocks to ease the burden of the testing process[15]

Non-scripted Tests are when the output of the code is not predefined or has an expected outcome unlike Scripted Tests. The tests dynamically explore the code's behavior seeking out any possible errors that may or may not have been seen through scripted code blocks. Its aim is to see unexplored or seek unexpected behaviors of a system, cases, or scenarios that might not have been anticipated by the development team.[15]

There are many ways to test Non-Scripted tests in Jest, one way is to generate random data and pass it through the function you are trying to test. This way we can see if there would be an appropriate behavior that will be done by a function call or rather an unexpected behavior we can use a library that is faker.js which generates random data which is obviously fake but to see whether the system would generate an unexpected output. We can also utilize 'expect' methods of jest where there would be a parameter and the parameter is the expected output of the code where it will check to see if the randomized data given will also give the same output. Non-Scripted tests on jest can be run by using randomized data to see if there would be an unexpected behavior of a system and utilization of other libraries

such as faker.js can be helpful in creating those randomized data.

3.4 Static Versus Dynamic Tests

Static Tests are when the documentation of the program and other related stuff like the code and program itself is being inspected. Although this part may not necessarily be executing the program code. Static Testing is focused on checking the code, and documentation before being able to run, It's main goal is to ind flaws and bugs to the system before it's is going to be tested for running with this type of test it is much easier to find the flaws and errors of both the program and documentation and eliminates the possibility of of other errors occurring before testing the program itself.

Files and systems subject to static testing may vary per system but it has some common things to test.It's common for code, design documents and requirements to be static tested before the software is run to find errors. Anything that relates to functional requirements can also be checked. More specifically, the process will involve reviewing written materials that provide a wider view of the tested software application as a whole.

Static Tests can be done in a few ways. There are some things that static tests can be of the following types which may be requirement specifications,design documents, user documents, webpage content, source code, test cases, test data and test scripts, specification and matrix documents.

Static Testing has different techniques as well and it targets specific things in the program such as the documentation to the code itself these techniques may target a specific area in order to ensure there are no errors in that area of the program. These testing techniques are some of the following. Informal Reviews Reviews the documentation and can give feedback about the document regarding any errors to the document. Walkthroughs are where the most knowledgeable of the team gives an explanation to the system and gives a demonstration of the said system. Technical Reviews is where the team inspects and checks whether the system conforms or follows the specifications and requirements of the projects and this can consider test plans, strategy plan, and specification documents for this type of static testing. Code reviews are done in which the team inspects the code but not necessarily runs the code but to see if there are any errors such as syntax or a violation to coding standards which may cause confusion for other team members. And lastly inspection where the team inspects if there are any errors at all they have a procedure and a checklist to see if there are any errors and if there are informs both the developers, and testers to rectify the situation

These tests have some benefits and drawbacks, One of the benefits is that it prevents any other errors from ever occurring once the actual testing of the system is done, these can save a lot of time and money for the developers as this can be resolved before the actual execution of the system. Another Benefit is that it is also cost effective, Errors found in static tests are much cheaper to fix rather than the ones found in dynamic testing, it greatly reduces the development, maintenance, organization, and overall costs of the project. Another is that it may find errors that cannot be found in a dynamic test and also may be able to fix syntax errors and other fundamental coding standards such as repeated functions, long methods, and such. This overall increases as well the productivity of every member of the team. However, there are also drawbacks to this approach, This also depends on the skills of the

reviewer of the code, it depends on the reviewer's skills if they may be able to uncover all the coding errors or documentation errors and that is another problem encountered in static testing it may not uncover all the problems that may arise. Another problem that is related to both problems said earlier is it may be time consuming as all other problems may arise. It may take much more time to find all errors possible and therefore might eliminate the cost efficiency advantage of the static testing.

Dynamic Testing on the other hand is a testing type in which it tests the program dynamically. It tests the program with dynamic or random generated testing data for the reason to see if the system will cause an error or even have a dynamic behavior to it. It mainly has dynamic input values and output values to be tested. It confirms if the system is indeed working as intended and conforms to the requirements given to by the client, it involves the actual testing of the program and executing dynamic data to validate the output and check for errors of the program, it can be slightly complex rather than the static testing and can have more realistic results rather than that of static testing. There are two main techniques implemented in dynamic testing which is white box and black box testing. White box again is a clear box testing where the tester knows the code clearly and knows what are the inputs and outputs of the test code. While for the black box they will test it without the knowledge of the code itself, they will test the system without the knowledge of the code and check if it conforms to the requirements given to by the client. There are many benefits to dynamic testing and there are also some drawbacks. The benefits may be that it can see if there are any errors in the runtime environment like maybe performance bottlenecks, errors in outputs, memory leaks and more. Another advantage is that it can verify and assess the integration of the modules and the reliability of the system. With dynamic testing it can check whether the codes are interacting with each other and can be run on stressful conditions such as accessing the web with multiple clients and such and checks the quality of life with the system for each and every user. It also has some drawback such as it may be very challenging for the user and the developer if there are any errors that are revealed in this type of testing it may also be time consuming since the error may be complex and in case of large systems this may cause a bottleneck in the system, another is that it requires effort from the developer and tester to fix errors that may occur and that is another disadvantage since it may not cover all scenarios possible due to time constraints.

In the context of Jest, static testing is done through the intervention of a professional or a tester of the code and maybe even the project manager, they can check for any errors in the code and check the documentation as well. As for dynamic testing, there are many tools available in jest for testing the code to see if there are any errors and there are many external libraries such as faker.js and others that may be able to help with the generation of data and scenarios to see if the system is working as intended.

4 UNIT AND INTEGRATION TESTING

Unit and Integration Testing are two very different concepts in Software Engineering but fall under the same category of the testing phase. Unit Testing is testing a module of the program separately from the whole system to test if there are any defects with that module. The testing may also be done with an individual function or procedure. This may be done by a developer

and verifies if there are any errors with the module itself.

Integration Testing is the process of testing the interface between two software units or modules, the focus being the correctness of the interface. One of its aims is to see if there's any fault in the interaction between the two integrated units. Once the unit Testing has been done successfully it will integrate the two modules together to see if the package is still working as intended.

4.1 Functional Analysis and Use Cases

Functional Analysis pertains to identifying factors that influence the behavior of a system. These can be used to improve the design of a system, improve problem solving, and overall improve the system. It may also describe it as a systematic process of identifying, describing, and relating functions of a system and how it must coexist with each other in order for the system to work successfully. But it does now address how well these systems will perform. The early stages of functional analysis deals with how top level functions need to be performed in the system, what functions need to be performed, how often they need to be performed and under what conditions.

We need to do functional analysis in order so that all the functions included in a system must meet the requirement specifications, it is required for subsequent requirements allocation, in essentiality it is describing what must be done and not how to do it we decouple requirements from implementation which leaves the it unbiased at all times.

Use Cases is a concept in which it describes how a system can be used to achieve specific goals or tasks. It also outlines how users and a system interact with each other. It can have scenarios in which it is a successful output or one with a failed output.

In the context of Jest there are ways in which we can implement use cases and functional analysis. These ways can be done by the concept of Unit Testing and Integration Testing where each module will be tested individually before being integrated into a single system unit to test if it still works and use real case scenarios to check if it will be viable for the real world. Use Cases may be able to help in the collection of data and results in functional analysis and there are a lot of functions that can be used by testers in order to help in the collection of data in the analysis. Jest Test hooks (i.e beforeEach() and afterEach()) allows for a controlled testing environment in which the test has random data generated in which the collected data will not affect the system in any way creating a clean setup and teardown of testing environment which the results can be then collected and also not affect the conditions of the function. Another way is the use of expect() function in which the test expects this type of output given test variables and also parameterized testing where the program is tested with different conditions and variables where there is an expected output and the result collected should match the expected output. These are some of the ways where jest can be able to use use cases and do functional analysis.

4.2 Exception Testing

Exception handling (EH) is a forward-error recovery technique used to improve software robustness [9]. An exception models an abnormal situation detected at run time that disrupts the normal control flow of a program [10]. When this happens, the EH mechanism deviates the normal control flow to the abnormal

(exceptional) control flow to deal with such a situation. Mainstream programming languages (e.g., Java, Python, and C#) provide built-in facilities to structure the exceptional control flow using proper constructs to specify, in the source code, where exceptions can be raised, propagated, and properly handled [11].

Recent studies have investigated the relationship between EH code and software maintainability[12], evolvability [13], architectural erosion [14], robustness [11], bug appearance [15], and defect-proneness [16].

These studies have shown that the effectiveness of EH code is directly linked to the overall software quality [17][18]. To ensure and assess the EH code, developers make use of software testing, which, in this context, is referred to as EH testing [19][20].

In exception testing, robust error handling is essential for managing unexpected issues that may arise during software operation. By identifying and addressing potential errors proactively, testers ensure the system remains resilient to crashes, data corruption, and security breaches [21]. Effective exception handling not only strengthens the reliability of the software but also enhances its overall stability and user experience, reducing risks in real-world deployment scenarios.

In Javascript, there are two ways in which we can implore Exception handling which is the try-catch, throw, and finally. These exceptions can be tested through Jest, Jest has its own ways to test exceptions in Javascript code Jest has two library functions which helps in testing errors which is the toThrow() and toThrowError(). To demonstrate this, let's say you have a code in Javascript where it's supposed to throw an error. How are you supposed to do this? In Jest the testing environment can intentionally create an error to see if the error does get invoked once it recognizes an error in the inputs of the testing area. You can write the tests with an error like this:

```
function divide(a, b) {
  if (b === 0) {
    throw new Error('Division by zero');
  }
  return a / b;
}

test('throws an error when dividing by zero', () => {
  expect(() => divide(10, 0)).toThrow('Division by zero');
});
```

Figure 10: *toThrow sample implementation*

That is one of the ways to implement and test an exception through Jest, but it is also important that we need to verify those exceptions when thrown, that is when the toThrowError() function works. There may be multiple errors thrown in a test and Jest can define particular error types by throwing the toThrowError() function. It verifies whether an instance of the error exists or not. It guarantees that the type of error is being thrown in the system.

```
class CustomError extends Error {}

function riskyOperation() {
  throw new CustomError('Something went wrong');
}

test('throws a CustomError', () => {
  expect(() => riskyOperation()).toThrowError(CustomError);
});
```

Figure 11: *toThrowError sample implementation*

In Javascript there are asynchronous code in which the program can run programs or functions in the background without disrupting the current state of the system and we can also test that in Jest. To test that one must know when to reject a promise with the right

kind of exception. In jest you can do that by adding a reject variable in the toThrow() method an example of the implementation can be seen below:

```
async function fetchData() {
  throw new Error('Network error');
}

test('throws an error when fetching data fails', async () => {
  await expect(fetchData()).rejects.toThrow('Network error');
});
```

Figure 12: *Async exception test implementation*

Note there are some mistakes that may be made when implementing these exception codes. Here are some of the notes we can give to avoid those mistakes. One is forgetting to wrap the function call in an arrow function. Jest requires the function call to be wrapped in an arrow function for the toThrow() and toThrowError() function to work properly. Another may be not specifying the error type properly, we need to be specific as possible when trying to test exception types correctly when using the toThowError(), we must be able to identify the proper error, otherwise the test may pass it as any other error and not the specific one you were looking for. Lastly when testing an asynchronous code, make certain that you use the await with rejects. toThrow. A common mistake here is actually forgetting 'await' What this does is that the test may pass, yet, the rejected promise is not checked.[30]

In conclusion, exception testing in Jest is a fundamental part and crucial part in testing whether your code is reliable and robust. Learning the functions of Jest such as the toThrow() and toThrowError() function calls when to use them and how to use them we can be able to ensure that the errors being thrown when the system has occurred an error is correct and not any random error has been thrown in the system.

4.3 Equivalence Partitioning

Testing is a very crucial process in software development, but it may take a lot of time considering manpower and when you are testing the system manually. In today's world that is not a problem anymore as there are multiple testing tools that have a feature called test automation. This helps development teams test and execute their code with a lot of other possible scenarios at a moment's time and helps accelerate the development process. However, the effectiveness and correctness of the testing greatly depends on proper testing techniques. With a concise approach to automated testing we can set some coverage of the test in the testing environment that may note some important issues when faced in automated testing, or may be inefficient tests that cannot be completed due to it defeating the purpose of automation. Equivalence partitioning is a strategy built for this. It is a black box software testing technique where it optimizes the number of test cases to an efficient one but still getting the desired and accurate result possible. It's concept is built allowing testers to classify their test suites and their partitions of data into classifying them according to how similar the behavior will be when the input was processed and an output will be displayed. This way, the tester can expect to have a series of errors while the redundancy of the errors to be minimal. This is crucial especially for testing a software that is in development since it can reveal things that the software might not be able to handle. Equivalence partitioning is a great testing technique that significantly enhances efficiency and effectiveness. By dividing the input into

partitions according to class functions we can create a tester that is smaller in size but yields the same comprehensive data as the large test case.

This approach maximizes test coverage, it reduces the time of testing, and increases the possibility of uncovering defects within the system, reusability of code, and ultimately makes the testing much easier. Equivalence partitioning can also uncover boundaries that the system may not have accounted for and helps in detecting errors early before actual testing with end-users.

Although it may also have disadvantages such as this is limited to black-box testing, it is not ideal for complex logic or algorithms and can miss potential defects of the system. Overall Equivalence partitioning is a very helpful technique in testing systems breaking down a system and its test cases into smaller pieces and creating test cases for those functions, this approach can give you a definite result to any possible defects in the system. In the context of Jest this can be done with the help of external tools such as external libraries and equivalence tools. We must first identify the parameters needed to be used in the test case and then define the equivalence classes, here you can use tools such as Test Case Generator, Equivalence Partitioning Tool, or Equivalence Class Partitioning Tool to generate the equivalence class according to what type of test you are going to be doing. Next is to choose which test case you are going to be testing in your selected field of test data and with that you can now test these cases on Jest. It may need a lot of work and other external resources available to your disposal but it is possible to do with Jest just as not as comprehensive as other testing tools.

4.4 Boundary Value Analysis

Boundary Value Testing (BVT) is another black box type of testing that is used to test the limitations of a system. BVT test cases are generated by using extremities of the input domain, such as what is the minimum value and maximum values given inside/outside values, and such with the goal in mind having to be testing the status of the System given these values in the boundary.[6] There are four forms of boundary value testing which is to be the normal BVT which tests only inside values of the domain and typical and common values Robust Normal BVT an extension to the normal approach this time considering as well the outside values. Worst-case Value Testing where the system is run with the worst case input variable in each scenario where the program might fail and the Robust Worst Case Scenario Testing with a much deeper test of approaching worst case scenario variable inputs.[6]
There are many advantages of Boundary Value Testing, one of which is that it can test where the limitations of the system can be hence the name of the test it also provides the maximum coverage with the minimal amount of effort involved in the test. It also covers or finds any defects on the edges of values in the input domain. Additionally it is a black box testing technique so there is no need to require the knowledge of the code and you can go simply straight to testing. However, there are also disadvantages. There might a need to find the boundary of a system to be able to do the testing process and processing a lot of data is really hard so partitioning of the said data may be necessary in order to get a concise result of the test, another problem that might occur is the validation of results which may take time since there is a need to define the boundaries and each boundary there has to be expected results and

outcomes which may take time to validate and see if there are any errors that are present.
This type of testing has many real world applications, including finance, construction, healthcare, and transportation. For example in the finance industry, Boundary Value Testing can be done on financial activities such as computation for tax and for the case of healthcare it can be used as a measurement tool to gauge a patient's diagnosis and prescribed treatment for the patient.

Boundary value testing in the context of Jest can be done in many ways. We can use test.each() and create scenarios in which we can test the minimum values and maximum values of a system where it might cause the system to create an error. We can also use the test blocks such as 'test' or 'it' blocks for testing the Boundary Value Tests cases. Additionally we can use the concept of Equivalence Partitioning in order to create a much more detailed and robust result of the test case involved.[6]

4.5 Decision Tables

Decision table testing is a type of black box testing technique where it breaks down a complex logic into a much more simple, clear and manageable format. It uses a table structure to format and define various input values and their corresponding expected outcomes. This approach ensures systematic test coverage and simplifies the testing process for complex systems. The decision table looks more of a grid which shows various inputs and the actions or corresponding expected outputs mimicking the behavior of an "if-then" statement making a clear visualization of the decision making process system. It has a few various components such as the input or the conditions of the statement, the expected outcome or the results of each input, rules explaining the relationship between the input and output values, and a legend giving information about the three variables the input, output, and rules.

This method of software testing is used to display and examine business decisions and rules and their interactions in a table grid. This is very helpful especially when trying to find results with varying outputs from different inputs. It helps in finding out the condition and decisions or actions from a combination of inputs ensuring a wide comprehensive coverage of results that can reveal any errors in the system.[link]
There are advantages to this approach such as having a systematic coverage where the tests tries all possible combination of inputs to create varying outputs, The simplification of a complex algorithm, It makes it also easier to trace if there is any error in the algorithm, It may also identify any missing logic if there are any due to the complex logic being broken down into a simpler version and efficient testing where redundant test cases may be reduced or eliminated due to having multiple combinations of test cases possible.

Jest has an implementation of these decision tables. One of the libraries that can be used is the it.each() function where the scope of this test in jest may be limited; here are some of the implementations available. We can use it.each in jest as a 2D array of variables this is a possible approach but it lacks in readability not helping you navigate where the input is going and or what it will be doing to the input [21]

```
const { extractUrls } = require("../src/extractUrls");

describe("extractUrls (Table-driven tests array format)", () => {
  it.each([
    ["Go to https://knowledge.com", ["https://knowledge.com"]],
    [
      "Go to https://knowledge.com and https://world.org",
      ["https://knowledge.com", "https://world.org"],
    ],
    ["Go to http://knowledge.com", ["http://knowledge.com"]],
    ["Go to google.com", ["google.com"]],
  ])("extractUrls(%s)", (message, expectedUrls) => {
    const extractedUrls = extractUrls(message);

    expect(extractedUrls).toEqual(expectedUrls);
  });
});
```

***Figure 13**: it.each 2d array implementation*

We can also create a decision table using a table of an array of objects using again it.each.

```
const { extractUrls } = require("../src/extractUrls");

describe("extractUrls (Table-driven tests object format)", () => {
  it.each([
    {
      message: "Go to https://knowledge.com",
      expectedUrls: ["https://knowledge.com"],
    },
    {
      message: "Go to https://knowledge.com and https://world.org",
      expectedUrls: ["https://knowledge.com", "https://world.org"],
    },
    {
      message: "Go to http://knowledge.com",
      expectedUrls: ["http://knowledge.com"],
    },
    {
      message: "Go to google.com",
      expectedUrls: ["google.com"],
    },
  ])(
    "should extract urls $expectedUrls from message $message",
    ({ message, expectedUrls }) => {
      const extractedUrls = extractUrls(message);

      expect(extractedUrls).toEqual(expectedUrls);
    }
  );
});
```

***Figure 14.** it.each array of objects*

This approach eliminates the problems with the other implementation regarding poor readability of the code giving a clearer picture of what is the input and where it might be going and also readability of variables involved.

Another way is having it as a tagged template literal.

```
const { extractUrls } = require("../src/extractUrls");

describe("extractUrls (Table-driven tests table format)", () => {
  it.each`
    message                                              | expectedUrls
    ${"Go to https://knowledge.com"}                     | ${["https://knowledge.com"]}
    ${"Go to https://knowledge.com and https://world.org"} | ${["https://knowledge.com", "https://wo
    ${"Go to http://knowledge.com"}                      | ${["http://knowledge.com"]}
    ${"Go to google.com"}                                | ${["google.com"]}
  `(
    "should extract urls $expectedUrls from message $message",
    ({ message, expectedUrls }) => {
      const extractedUrls = extractUrls(message);

      expect(extractedUrls).toEqual(expectedUrls);
    }
  );
});
```

Figure 15. *tagged template literal implementation*

This approach is the best approach as far as readability goes and where it resembles the closest to a decision table. The number of variables available is much less and more compact helping with the readability of the code.

In conclusion, decision tables can be able to create a comprehensive output and give different non-redundant outputs and can be a good testing type to check for any errors that may occur regarding the combination of conditions and outputs available. This type of testing is also available in Jest but it has a limited implementation and can be used in it.each test case[21]

5 SYSTEM AND USER ACCEPTANCE TESTING

System testing is a type of software testing that validates whether the fully integrated hardware and software meet specified requirements and align with the project's goals [22]. It ensures there are no irregularities in the integration of subsystems within the system and evaluates end-to-end specifications. This testing is conducted after unit and integration testing, making it the first phase where the entire system is tested as a whole [23].

System testing is particularly used to analyze the end-to-end flow of an application. Testers navigate all required modules, verify that the end features work as expected, and test the product as an entire system [24].

In the broader context of software validation, System Testing complements User Acceptance Testing (UAT). User Acceptance Testing (UAT) is a critical testing method involving direct interaction between the user and the system to verify that the system meets the user's needs and expectations [25]. While system testing verifies that the system operates correctly as a whole and meets technical and functional requirements, UAT evaluates whether the system satisfies user needs and expectations in a real-world operational context [26]. UAT is typically performed after system testing as the final step before release, verifying that the software meets user-defined acceptance criteria agreed upon by developers and users. UAT produces a test results report, which serves as the basis for determining whether the system meets the user's needs (Ganesh et al., 2014). It acts as the final step in the development process to ensure that the software is ready for operational release and can effectively support the intended use case.

Once these criteria are fulfilled, the software is considered ready for deployment. Together, system testing and UAT ensure that the software is robust, functional, and user-centric.

In these series of tests, a GitHub repository (see Appendix D) will be used, which is an e-commerce product store application created using the MERN stack (MongoDB, Express, React, and Node.js). This serves as the software which will be tested to ensure that it meets the specified requirements and aligns with the project's goals. The system testing process will validate the integration of hardware and software, checking that all subsystems work together seamlessly.

5.1 Cross-Feature Testing

Cross-feature testing is a type of software testing aimed at identifying bugs that emerge only when multiple features of a product interact, but remain undetected when each feature is tested in isolation **[6]**. This area of testing is particularly challenging due to the vast range of possible interactions and the need for in-depth understanding of all features involved. Consequently, cross-feature testing requires a larger number of test cases and introduces added complexity in designing and executing these tests. In this discussion, we examine the significance of cross-feature testing, particularly in the context of the e-commerce product store, and explore its role in validating software quality and reliability.

In modern software systems, especially those built with complex frameworks like MERN (MongoDB, Express, React, Node.js), multiple features and subsystems often interact [27]. While unit testing and integration testing focus on validating individual components or their interactions in isolation, cross-feature testing ensures that the integration of these components does not introduce unintended behaviors.

In the case of the product store, cross-feature testing is crucial for validating the system's integrity across various operations, such as product creation, updating, fetching, and deletion. These actions, while functional when tested in isolation, interact with one another in ways that could result in bugs if not properly integrated. For instance, creating and fetching a product should update the store's state consistently, while deleting a product must ensure that the correct product is removed without disrupting the rest of the product list.

5.2 Cause-Effect Graphing

A cause-effect graph is a graphical notation for describing logical relationships between causes and effects in a software specification [28]. A cause is a binary-valued atomic sentence in the specification and represents an input value (or state or event), an effect is also a binary-valued atomic sentence representing an output state or action. For each feature in the software specifications, we isolate causes that influence the feature's behavior. Then, we derive a graphical representation of the cause and effect relationships by connecting the causes and effects with Boolean operators.

In the context of the product store, where the functionality revolves around managing a list of products, ensuring the correctness and usability of the core features is critical.

To apply cause-effect graphing effectively, we identify key input conditions (causes) and their resulting behaviors (effects) for core operations:

Key Causes (Input Conditions)
C1: Valid product data provided (all fields: name, image, price).
C2: Unique product name (no duplicate name in products).
C3: API responds successfully (e.g., data.success = true).
C4: Valid product ID (pid) provided for delete/update.

Key Effects (Output Results)
E1: Product is successfully added to the products list.
E2: Product creation fails (validation or API error).
E3: Product is successfully deleted from the products list.
E4: Product update is reflected in the products list.

The following table maps causes to their effects using Boolean logic.

Table 1. *Cause-Effect Mapping*

Causes	Effects	Scenario
A. C1 AND C2 AND C3	E1	Valid products are created successfully.
B. NOT C1 OR NOT C2	E2	Product creation fails due to invalid input.
C. NOT C3	E2	Product creation fails due to API error.
D. C4 AND C3 (DELETE)	E3	Product is deleted successfully.
E. C4 AND C3 (UPDATE)	E4	Product is updated successfully.

Below is a simplified graphical representation of the logical relationship between C1, C2, C3, and E1.

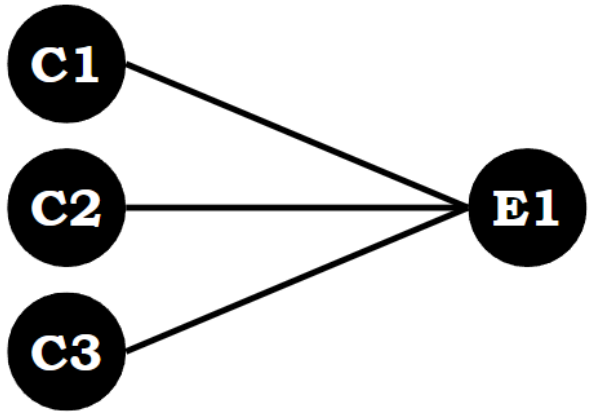


Figure 16. *Cause-Effect Graph for A.*

The cause-effect mapping leads to the following structured test cases:

Table 2. *Test Cases*

Test Case	Condition	Expected Result
TC1	Valid product data (C1, C2, C3)	E1: Product added to products.
TC2	Missing fields (NOT C1)	E2: Product creation fails.
TC3	Duplicate product name (NOT C2)	E2: Product creation fails.
TC4	API fails (NOT C3)	E2: Product creation fails.
TC5	Valid delete request (C4, C3)	E3: Product removed.
TC6	Valid update request (C4, C3)	E4: Product updated.

The created test cases focus on validating the core functionalities of the product store by using the cause-effect mapping derived from the input conditions (causes) and expected output results (effects).

Test Case 1 (TC1) tests the scenario where valid product data is provided, ensuring that the product is successfully added to the products list, reflecting the effect of E1. Test Case 2 (TC2) simulates the situation where required product fields are missing (C1), leading to a failure in product creation, thus triggering E2. In Test Case 3 (TC3), the product creation fails if a duplicate product name is detected (C2), resulting in the effect E2. Test Case 4 (TC4) simulates a situation where the API response fails (C3), causing product creation to fail, which again leads to E2.

Test Case 5 (TC5) tests the functionality of product deletion by sending a valid delete request (C4, C3), ensuring the product is successfully removed from the list, corresponding to the effect E3. Lastly, Test Case 6 (TC6) evaluates the functionality of updating a product. If a valid update request is made (C4, C3), the product should be successfully updated, and the effect E4 is expected.

These test cases cover the essential paths of the system, ensuring that the product store behaves correctly under different conditions by confirming that the appropriate actions (addition, failure, deletion, or update) occur based on the input scenarios.

5.3 Scenario Testing

Scenario testing is a structured and intentional approach in software testing that aims to replicate real-world situations, user behaviors, and the varied interactions that a software application may encounter once it is deployed. The main objective of this testing method is to simulate actual conditions to evaluate how the software behaves under those circumstances. This type of testing helps to uncover potential bugs, weaknesses, or usability issues, ensuring the software aligns with user expectations and performs effectively when subjected to real-world conditions. The process is crucial in validating the software's ability to meet business requirements, user needs, and functional goals, offering confidence in its robustness post-launch.

According to Kaner [29], scenario testing possesses several key attributes that enhance its value in software evaluation. One of the key strengths of scenario testing is that it typically presents test cases as

narratives or stories, which mirror practical and relatable use cases. This approach helps stakeholders, including developers, business owners, and end-users, better understand the expected behavior of the software, thus fostering clearer communication and ensuring alignment with user expectations. Additionally, scenario tests engage stakeholders by reflecting their interests and requirements, making the testing process more inclusive and relevant to those invested in the software's success. This involvement not only boosts the credibility of the tests but also provides a deeper understanding of how the system is expected to behave in real-world conditions.

The focus on realistic use cases is another crucial characteristic of scenario testing. By testing the software under conditions that closely resemble actual user interactions, the tests increase stakeholder trust in the product and in the testing process itself [84]. This trust is vital for ensuring that the software is reliable and capable of handling real-world situations, making scenario testing a powerful tool for validating product functionality and usability. Furthermore, scenario tests consider multiple inputs and conditions, which enables testers to examine how the software responds to various combinations of factors. This comprehensive approach ensures that no aspect of the software's performance is overlooked, providing a thorough evaluation of its capability to handle diverse scenarios.

While scenario tests can be complex in nature, they are designed to be straightforward in terms of evaluation. By using realistic conditions and clear, relatable examples, testers can quickly assess whether the software is meeting its requirements and whether it is likely to perform well once in the hands of users. The simplicity of evaluation allows for rapid decision-making, which is critical for addressing issues promptly and iterating on the software development process as needed.

5.4 Usability Testing

Usability testing refers to methods in which users engage with a product or system in a structured environment, performing specific tasks designed to achieve a particular goal. During this process, data regarding user behavior is collected to assess the system's effectiveness and usability [30].

Usability testing helps identify design flaws that may have been missed otherwise. By observing how test users interact with the product while attempting tasks, you gain valuable insights into its effectiveness. These insights can then be used to enhance and refine the design. According to the Interaction Design Foundation [31], whenever you run a usability test, your chief objectives are to:

1. Determine whether testers can complete tasks successfully and independently.
2. Assess their performance and mental state as they try to complete tasks, to see how well your design works.
3. See how much users enjoy using it.
4. Identify problems and their severity.
5. Find solutions.

In Jest, usability testing can be taken a step further by leveraging the framework's powerful automation capabilities to simulate real-world user interactions. While it is typically used for unit and integration testing, it can also play a crucial role in usability testing. By using Jest in combination with *fireEvent* from the React Testing Library, developers can

simulate user actions, such as button clicks, form submissions, and navigation, to test how the application responds to various inputs in a controlled environment.

As technology continues to evolve, the role of usability testing becomes increasingly significant. With the growing complexity of user interfaces and the diversity of user needs, the insights gained from usability testing will be crucial in shaping the next generation of products. By embracing a user-centered design approach, companies can not only meet the functional requirements of their users but also enhance their engagement and satisfaction. Moving forward, the integration of more advanced usability testing methods, such as remote testing and AI-driven user behavior analysis [32], is likely to revolutionize the development process, further cementing usability testing as an indispensable tool in product design and innovation.

6 SPECIAL TESTS

Special tests in software testing are specifically different, unique methods used to test certain functionalities or characteristics of a software application that would go beyond the standard testing. Most of the time, these tests are designed to address specific problems, such as how the system performs under extreme conditions, the interaction with other systems, or the user experience in various environments [33].

Specialized tests are required for applications that meet special needs, for instance, for applications deployed in critical environments. In such deployments, the failure to meet specific standards may lead to major problems. For instance, recovery testing should ensure there is recovery capability from failure while interoperability testing should confirm compatibility with other systems and services. Data migration testing has to be correct to ensure proper data transfer between different systems; otherwise, some data would be lost or corrupted.

Special tests come into play as software grows more complex, ensuring the robustness, security, and efficiency of it under certain conditions that are difficult to achieve by standard testing methods.

6.1 Smoke Tests

A smoke test is the first line of verification while the software is under development [34], similar to a pre-flight check on an airplane before takeoff. The testing approach made sure that software's most basic fundamentals are working properly so detailed testing isn't wasted on time. For example, testing in an online shopping website or online marketplace involves systematic validation of whether or not users can perform basic but crucial tasks-from account creation and product browsing to purchasing with various payment means. It's especially meaningful to consider its practical impact: without it, one might waste an entire testing team of 10 people 8 hours each (that is, 80 hours of work) testing a fundamentally broken software version. It is because teams can avoid such wasteful situations if they run a quick smoke test that essentially takes about 60 minutes at first. The cycle of process starts when developers deliver the software to the Quality Assurance team, who then run these basic checks. If the software passes such necessary tests for functions, then it will move forward for detailed testing; otherwise, it sends it back to the developers for its fixes. This approach can be done either in manual mode (hands-on) or through automation tools, which run recorded test cases to ensure software stability.

```
describe('CoinLore API Smoke Tests', () => {
    test('should successfully connect to global markets endpoint', async () => {
      const response = await fetch('https://api.coinlore.net/api/global/');
      expect(response.status).toBe(200);
      const data = await response.json();
      expect(data).toBeDefined();
    });

    test('should retrieve Bitcoin data correctly', async () => {
      const response = await fetch('https://api.coinlore.net/api/ticker/?id=90');
      expect(response.status).toBe(200);
      const data = await response.json();
      expect(data[0]).toHaveProperty('symbol', 'BTC');
      expect(data[0]).toHaveProperty('name', 'Bitcoin');
    });
  });
```

Figure 17. *Smoke Test sample using CoinLore API*

Jest offers strong functionality for rapid verification of critical system functionalities. Jest's describe and test blocks lets developers write efficient smoke test suites that verify basic features of an application and core functionality. The fact that this framework allows running tests in parallel makes this process particularly efficient, where it enables development teams to identify basic issues rapidly before proceeding with further testing phases [35].

6.2 Performance Tests

Performance Testing is an all-inclusive term that incorporates different testing techniques designed to measure system behavior and set performance standards [36]. Here, because defect detection is not always of interest in performance testing, key performance indicators such as throughput, response time, speed, stability, and resource utilization KPI are measured; thus, the insights developed here provide a good foundation on scalability and resource usage and much more related aspects for understanding the overall reliability of a system. This test method offers a basis of understanding the behavior of a system with regard to various conditions and assists in establishing benchmark performance.

```
async function someOperation() {
    return new Promise((resolve) => setTimeout(resolve, 500)); // simulate 500ms
  }

describe('Performance Test', () => {
    test('response time should be under threshold', async () => {
      const startTime = performance.now();
      await someOperation();
      const endTime = performance.now();
      const executionTime = endTime - startTime;
      expect(executionTime).toBeLessThan(1000); // 1000ms threshold
    });

    test('multiple operations', async () => {
      jest.useFakeTimers();
      const operation = jest.fn();
      setTimeout(operation, 1000);
      jest.runAllTimers();
      expect(operation).toBeCalled();
    });
  });
```

Figure 18. *Performance Test sample using Timer Mocks*

Although Jest is not a performance testing tool, it carries some important features that measure and monitor system performances. The feature Timer Mocks in Jest enables developers to measure execution times, create delays, and evaluate time-sensitive operations practically [35]. This capability is very useful especially when testing operations require precise time measurements or performance benchmarks.

6.3 Load/Stress Tests

Load testing verifies system behavior under anticipated normal conditions by performing a gradual increase in load to mimic typical user traffic and expected peak conditions [36]. Load tests are carried out close to the end of a project. It helps in specifying potential problems such as buffer overflow, memory leak, and bandwidth limitations while establishing an upper performance limit for every system component. Stress testing, is pushing the system purposely beyond its normal operational capacity in order to understand its behavior under extreme conditions like overloading the system, removing components, and conducting spike and soak tests bringing to the light some critical problems like memory leak, race conditions, and synchronization issues. Differences between approaches include the purpose that each is intended to serve: load testing verifies system behavior at expected levels, while stress testing intentionally tests system behavior at and above its operational limits to identify failure modes and recovery profiles.

```
describe('Basic Load Test', () => {
    test('handle multiple concurrent requests', async () => {
      // simulation of a mock async operation
      const someAsyncOperation = () => new Promise(resolve => {
        // simulate 100ms operation
        setTimeout(() => resolve({ status: 'success' }), 100);
      });

      // simulate concurrent requests
      const requests = Array(1000).fill().map(() => someAsyncOperation());

      const results = await Promise.all(requests);

      results.forEach(result => {
        // expect the status to be 'success' for each request
        expect(result.status).toBe('success');
      });
    });
  });
```

Figure 19. *Load Test sample using simulated async operation and concurrent requests*

Jest also provides the basics of load and stress testing, enabling it to check the system under various conditions. There is also a built-in retry mechanism and timeouts by Jest for detecting bottlenecks in performance and stability issues [35]. Specialized tools are however very likely to be needed when using load testing for heavier scenarios such as k6 and Artillery.

6.4 Reliability Testing

Reliability Testing is an important evaluation tool for products and systems, which assumes several different testing techniques to check and enhance product reliability [37]. This has gained tremendous importance because companies strive to improve their competitiveness through new products with upgraded functions and materials. It includes several specialized methods: HALT (Highly Accelerated Life Tests), RGT

(Reliability Growth Tests), HASS (High Acceleration Stress Tests), RDTs (Reliability Demonstration Tests), Reliability Acceptance Tests, Burn-in Tests, Built-in Self-Tests (BIST), ALTs (Accelerated Life Tests), and ADTs (Accelerated Degradation Tests).

These tests are important because reliability directly affects the quality of the product as well as that of the company in terms of productivity. Reliability testing determines the probability of a product working correctly under specified conditions during its designed lifetime [37]. Testing can either be under normal operating conditions which would yield closer to the real measurable metrics or by accelerated techniques to impose higher stress levels where results are produced fast and relatively less expensive.

```
describe('Reliability Test', () => {

  // simulating an operation that should be consistent
  const someOperation = () => {
    return 42;
  };

  // snapshot function for generated data
  const generateTestData = () => {
    return {
      id: 1,
      name: 'Test Item',
      value: 100,
      timestamp: new Date().toISOString()
    };
  };

  test('consistent output across multiple runs', () => {
    // run function multiple times and verify consistent output
    const result1 = someOperation();
    const result2 = someOperation();
    expect(result1).toEqual(result2);
  });

  test('snapshot test for data structure', () => {
    const data = generateTestData();
    // snapshot test to detect changes in data structure
    expect(data).toMatchSnapshot();
  });
});
```

Figure 20. *Reliability Test sample using Snapshot*

Jest really excels with its robust feature set that includes snapshot testing, and full assertion capabilities. Snapshot testing is a feature that is specifically helpful in detecting unexpected system behavior changes and ensuring that outputs are reliable across updates [35]. Jest's simulation of different environmental conditions helps to ensure that systems are reliable under a variety of scenarios.

6.5 Acceptance Tests

Acceptance Tests are the final validation phase in software development wherein actual end-users evaluate whether the product meets real-world business requirements. This is also known as User Acceptance Testing or UAT, but it takes place before the live system and is a certification stage on the verification of end-to-end business workflows. In acceptance testing, the users perform tasks within the corresponding testing environment with a focused objective of verifying whether the software will function as expected and meet organizational needs. This testing is spread over several test cycles over several days when the participants have to follow specific test scenarios and provide minute details of the outcomes. The prerequisites for this testing include completion of system integration testing, regression testing, and proper configuration of the test environment. Most organizations document the test results, tracking acceptance criteria, business impact levels, and pass/fail results. Acceptance tests identify gaps between the delivered solution and business expectations by testing each requirement as a whole. This can act as the last quality checkpoint before deployment. There is a growing need for proper planning, clear scope definition, and effective collaboration between business users and development teams to make these tests successful [38].

Jest provides an all-around system that makes verification of business requirements straightforward. The readable syntax and detailed report from the framework make it more accessible for stakeholders to understand the test results and verify the acceptance criteria. The support of async/await operations in the framework was especially helpful in testing complex business workflows [35].

7 TEST EXECUTION

Test execution in the context of software testing is the implementation of test scenarios to verify a system's functionality and performance. In a gist, it is conducting a practical search for bugs in the code by using test cases that are created during the previous stages of the development process [38]. For Jest, this process ensures that importance is given to discovering defects, validating behavior, and ensuring good software quality

7.1 Test Cycles

Test cycles can serve as methodologies for systematic executions of tests. The Arrange-Act-Assert pattern involves three steps as a way to structure test cases: Arrange, which involves setting up the test case; Act, which covers the main thing to be tested, including calling a function or method; and Assert, which involves the expected outcomes and should elicit some sort of response [39]. In the context of Jest, this pattern can be followed to structure tests efficiently.

The cycle begins with the Test Suite initialization, which involves importing modules, loading files from the local file system, configuring the test for all options, and defining lifecycle functions like the setup and teardown stages [40]. This stage typically involves setting up connections, importing modules, etc.

After initialization, Individual Test Preparation is done, where the development of testing conditions and preparation of mock objects, along with the generation of test data, are performed to simulate various software interaction scenarios. Thus, by creating properly isolated and properly defined testing scenarios, this phase ensures that each test could be executed independently, with controlled variables and clear

expectations. Jest provides mechanisms for test preparation through its lifecycle hooks and mocking features. This allows for the creation of controlled testing environments and prepare test data effectively. These preparation techniques are discussed in part 7.3, which covers Jest's approach to test data management.

The core part of software verification is represented by the Test Execution phase, in which individual tests are processed, assertions are applied, and developers compare expected results to the actual behavior of the software [38]. Each precise test case investigates specific functional aspects alongside logic validation and discovering potential defects or vulnerabilities. Jest executes tests using readable and intuitive syntax. The framework's capabilities, including implementation of the Arrange-Act-Assert pattern, will be demonstrated in Chapter 7.2 when discussing the approach to test new features.

After the execution of tests, Test Teardown makes sure that the environment is cleaned up, and any changes made to the states of the program are reverted back to avoid side effects. This process includes removing temporary resources, reverting modifications to configurations, and releasing system resources or allocations that were used during testing. The Teardown stage prevents potential interferences that may happen with different test scenarios and ensures the integrity of subsequent tests. Jest's teardown functionality makes sure that proper cleanup is done after test execution, preventing test pollution and maintaining isolation. These cleanup mechanisms are illustrated in part 7.2, where management of test environment is discussed and in part 7.3's coverage of test data handling.

Lastly, Result Reporting captures the outcomes of the tests transforming raw test data into insights about the quality and performance of the software being tested. This stage involves generating documentation that shows the outcomes of test executions, show passed and failed test cases, and provides information about any issues that may arise. This reporting process is not limited to simple pass or fail indicators, but it also includes in-depth analysis that helps developers understand potential problems, their specific locations or lines in the codebase, and their impacts on the overall system functionality. Jest offers reporting features that provide insights into results and execution status. The framework's reporting capabilities are discussed in part 7.4, which focuses on tracking and reporting test results.

7.2 Approaching a New Feature

The testing approach when introducing a new feature to an existing codebase must ensure that the feature being added integrates seamlessly without compromising the software's existing functionalities. This approach reduces the likelihood of defects slipping through, allowing for quicker issue resolution. Testing new features involves validating functionality against requirements, ensuring that it is compatible with existing components, and preparing for unexpected user behavior [41].

Testing frameworks such as Jest are critical in the current state of software testing. It helps developers in executing repeatable, efficient, and reliable tests. Jest can streamline isolated and automated testing to support rapid iterations while exerting minimal effort. The following steps outline a strategy for approaching a new feature using Jest:

Before writing any test cases, it is important to have a deep understanding of the feature's functionality, purpose, and expected behavior. A test case ensures

that the software operates as planned and that all of its features and functions perform correctly. Testers and developers typically create these tests to guarantee that the software meets the specified requirements and specifications [42]. Without understanding this, test cases may miss critical scenarios, leading to defects or misunderstandings with stakeholders' expectations. Once the requirements are understood, Jest can be used to translate these requirements into more structured and maintainable test cases:

```
test('should return error for invalid email format', () => {
    const invalidEmail = 'invalid@.com';
    const result = validateEmail(invalidEmail);
    expect(result).toBe(false);
});
```

Figure 21: *Jest Unit Test for Email Validation Function*

Jest's test() and it() methods allow developers to write clear and descriptive names for test cases. These names function as documentation for the test suite, emphasizing the purpose of each test case in natural language. This strength is important when collaborating on large projects since it ensures that each member of a team can easily understand the intent of the test without knowing the implementation details.

```
describe('New User Registration Feature', () : void => {
    test.each([
        ['valid email and password', 'user@example.com', 'password123', true],
        ['missing email', '', 'password123', false],
        ['invalid email format', 'user@.com', 'password123', false],
    ])('returns %s for input: %s, %s', async (desc, email, password, expected) : Promise<void> => {
        const result = await registerUser(email, password);
        expect(result.success).toBe(expected);
    });
});
```

Figure 22. *Jest Test Suite for New User Registration Feature with Data-Driven Testing*

Jest's each() method enables parameterized testing to reduce redundancy in tests. Instead of duplicating logic for similar cases, developers will have the ability to define multiple test cases in a single structure. This makes the test suite concise and maintainable while ensuring that the coverage is comprehensive.

Lastly, descriptive test case names and parameterized tests can be combined to ensure that all feature requirements are tested thoroughly. This minimizes the risk of oversight by specifically covering each condition stated in the requirements. In addition, Jest's automated testing capabilities can repeatedly run tests throughout the development cycle to catch regressions early.

Properly setting up the test environment is a crucial step to ensure consistent and reliable results. The test environment, sometimes referred to as a test bed, should be configured according to the needs of the software being tested. No matter the testing project, the test environment must be set up accordingly to ensure that the software operates correctly [43]. Jest provides a rich set of tools and excels in configuring test environments. It offers features allowing developers to simulate real-world conditions while having the ability to

control external dependencies. An elaboration of how Jest's features are elaborated as follows:

```
jest.mock('axios'); // mock API calls globally for the feature

describe('User Login Feature', () : void => {
    beforeEach(() : void => {
        jest.clearAllMocks(); // reset mocks in between tests
    });

    test('should return user data on successful login', async () : Promise<void> => {
        axios.post.mockResolvedValueOnce({
            data: { id: 1, name: 'John Doe' },
        });

        const userData = await loginUser('john@example.com', 'password123');
        expect(userData).toEqual({ id: 1, name: 'John Doe' });
    });

    test('should handle login failure gracefully', async () : Promise<void> => {
        axios.post.mockRejectedValueOnce(new Error('Invalid credentials'));

        const response = await loginUser('invalid@example.com', 'wrongpassword');
        expect(response.error).toBe('Invalid credentials');
    });
});
```

Figure 23. *Jest Test Suite for User Login Feature with Mocked API Calls*

Mocks (mock()) and spies (spyOn()) methods offer functionalities in Jest for simulating dependencies and interactions externally. These utilities allow developers to replace real implementations with mock versions, allowing the code to be isolated under tests. This is particularly useful for testing features that rely on APIs, databases, or other external systems.

```
// jest.config.js
module.exports = {
    // load global configurations
    setupFiles: ['<rootDir>/testSetup.js'],
};

// testSetup.js
jest.mock('axios'); // mock axios for all tests

// set global environment variables
process.env.API_BASE_URL = 'http://localhost:3000/api';
```

Figure 24. *Jest Configuration and Setup for Global Mocks and Environment Variables*

Jest allows global configurations with setupFiles or setupFilesAfterEnv. This allows configurations to be initialized before the test run, ensures consistency, and avoids repetitive setup in individual testing environments. For features requiring specific runtime configurations, setupFilesAfterEnv can be used to load additional settings right after Jest initializes.

```
beforeEach(() : void => {
    cart = new ShoppingCart();
});

afterEach(() : void => {
    cart.clear(); // clear the cart to reset state
});
```

Figure 25. *Jest Setup with beforeEach and afterEach for ShoppingCart Initialization and Cleanup*

Jest's lifecycle methods, such as beforeEach() and afterEach(), make sure that each test runs in an isolated

context. This prevents states from being residual, affecting subsequent tasks.

When introducing a new feature, regression testing makes sure that existing functionalities remain unaffected by the changes. In the context of Jest, regression testing helps in automation, integration with CI/CD pipelines, and test-case management to provide feedback on potential regressions.

Regression testing involves the repetition of previously executed test cases to confirm that the new feature has not introduced defects. This approach ensures that frequent updates and iterative development are common, especially in agile development and automated regression testing further accelerates this by minimizing manual interaction.

Jest plays a crucial role in regression testing through its test execution framework and automation features. As discussed in Part 9.4: Automation of Regression Testing, Jest's ability to integrate with CI tools allows teams to automate regression tests at every build stage. In addition, Jest's snapshot testing, highlighted in Part 9.4, allows the rendered output of components or functions to be captured and compared to future outputs against these baselines. Features like snapshot testing and seamless integration with CI tools testing frameworks having these features ensure stability by automatically identifying deviations in component behavior or system performance during regression tests [42].

Performing Code Coverage Analysis — As outlined in Part 3.1: Test Coverage and Testing Dimensions, coverage analysis ensures that all parts of the codebase are reached when being tested. By analyzing metrics such as statement, branch, function, and line coverage, developers can gain a view of how much their code is being tested. These metrics help pinpoint the areas that may not be properly tested. As Rahman & Mabood [49] state, the ability to measure coverage allows teams to ensure that their code is properly tested before it is deployed, preventing the risk of shipping untested or vulnerable code.

7.3 Test Data

Test data is an important component of software testing which helps validate the functionality, performance, and reliability of an application. Creating and managing test data is not just about generating random inputs, but it is about designing data that can uncover vulnerabilities, consider edge case scenarios, and ensure that the software performs as expected [46].

Jest handles test data through parameterized testing. It provides the each() method, which is perfect for running a single test case with multiple sets of inputs. This method helps ensure that different input combinations are properly tested without code duplication and is particularly useful when testing components or functions that contain the same logic for different inputs. This approach is important when validating functionality against broad data. In Part 7.2: Approaching a New Feature, parameterized tests in Jest were discussed as a means to eliminate redundancy. This type of testing makes sure that the edge cases and variations of input are evaluated.

Jest also offers mocking capabilities, such as the mock() method, which enables the simulation of external data sources, APIs, or databases. By mocking data, tests become isolated from external dependencies. In Part 7.2, Jest's mocking features facilitate the features reliant on external systems, which are discussed. This makes sure that tests are focused primarily on specific features without being affected by the availability or state of

external sources, helping to achieve faster and more stable test execution.

Dynamic data generation plays an important role when it comes to ensuring the resilience of a software application across various scenarios. With Jest, the generation and usage of dynamic data is made easier by applying data creation strategies, where developers can test components more effectively by simulating a range of possible inputs, including edge cases and large datasets. Testers who use test automation recognize the value of dynamic test data, as highlighted in the field of Test Data Management: “Test cases developed during an Agile sprint makes it possible for testers to quickly configure a Test Data Case containing instructions for generating real-time synthetic test data during test execution. This is the essence of Test Data Automation, which is the ability to model and design the data that you need and immediately deploy it into a test automation environment” [47].

```
function calculateTotalPrice(cartItems) {
    return cartItems.reduce((total, item) => total + item.price * item.quantity, 0
}

test('calculates total price of items in the cart correctly', () : void => {
    const cartItems : [{quantity: number, price: num... = [
        { name: 'Apple', price: 1.0, quantity: 3 },
        { name: 'Banana', price: 0.5, quantity: 6 },
        { name: 'Cherry', price: 2.0, quantity: 2 },
    ];

    const result = calculateTotalPrice(cartItems);

    expect(result).toBe(10.0); // hard-coded expected total price
});
```

Figure 26. *Jest Test for Calculating Total Price of Items in a Shopping Cart*

Hard-coded test data refers to manually writing values within test cases. These data values are typically simple and are directly embedded into the test code. In Jest, this type of data is useful when developers are testing certain cases where input data is known and consistent. For example, testing a function that performs calculations based on fixed numbers and values.

```
const faker = require('faker');

test('generates random user data', () => {
    const user = {
        name: faker.name.findName(),
        email: faker.internet.email(),
        address: faker.address.streetAddress(),
    };

    expect(user).toHaveProperty('name');
    expect(user).toHaveProperty('email');
    expect(user).toHaveProperty('address');
});
```

Figure 27. *Jest Test for Generating Random User Data Using Faker*

Generated test data refers to creating data dynamically during the execution of the tests. This is commonly achieved by using libraries that generate random values or realistic data. In the context of Jest, libraries like Faker or Chance can be integrated inorder to create dynamic test data. This approach is valuable for testing functions or components that require large data or a variety of inputs.

External data fixtures refer to predefined datasets stored outside of the test code, such as JSON files or static files. These data can be loaded into the test suite before or during test execution to provide consistent sets of inputs for testing. For Jest, this strategy can be used when dealing with realistic or large datasets that would be inconvenient to generate manually or programmatically. These external fixtures can be loaded using beforeEach() or setup functions in Jest.

7.4 Tracking and Reporting Results

Tracking and reporting results is a part of the QA process, which helps to ensure that the software meets the defined requirements and customer expectations . In Jest, there are various methods of reporting and tracking that can be utilized to monitor tests and detect potential issues early on. These tools allow teams to retrieve insights into test results, identify regressions, and improve the testing process overall.

```
PASS ./jest.test.js
  ✓ (6 ms)
  Scoped / Nested block
    ✓ (8 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.04 s, estimated 5 s
Ran all test suites.
```

Figure 28. *Jest Test Console Output Sample*

Jest provides a variety of reporting mechanisms to provide developers with feedback about their tests. These reports are crucial for understanding how tests perform and for detecting issues in the codebase. Jest provides detailed test results directly in the console. By default, Jest outputs a summary of all test results, showing whether each test has passed or failed. Additionally, Jest provides a simple color-coded output, making it easy to distinguish between passing and failing tests.

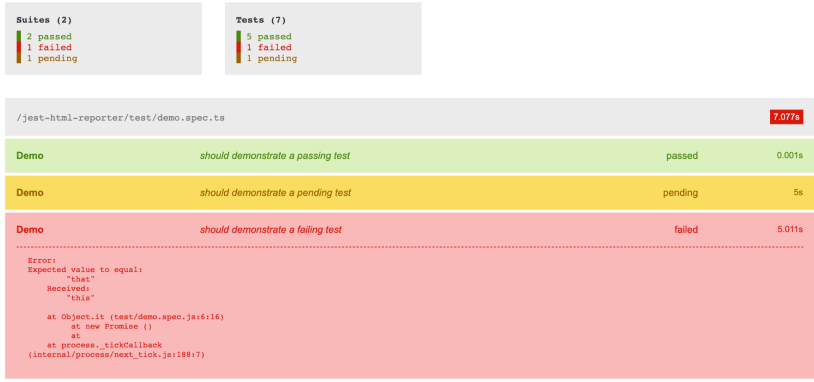


Figure 29. *Jest HTML Report Sample*

For a more user-friendly and visually pleasing output, Jest can generate HTML reports. These reports provide an overview of the test suite, and a breakdown of results, individual tests, and pass or fail statuses. HTML reports come in helpful when teams need a more structured view of test results useful when working with large codebases or sharing reports and results with non-developers. Jest supports integration with tools like jest-html-reporter that can output HTML reports from test results, as shown in Figure 29.

Jest can also generate test results in JSON format, which is useful when processing or integrating the results with other tools or CI/CD systems. JSON reports allow for further manipulation and program

parsing to automate the decision-making process, such as triggering other workflows or alerting developers of failing tests. In addition, and as discussed previously, Jest provides code coverage reports to help developers track how much of their code is being covered by tests. These reports can be displayed directly in the console or output to a separate file in several formats, such as text, HTML, or JSON.

Tracking the results of tests is as important as running them. They provide insights regarding the quality of the codebase and help identify trends, such as repeated failures, performance issues, or test flakiness. As Dhaliwal states, "Looking at the results of a single test ruin in isolation is better than nothing, but keeping a history of all your test runs can help you perform comparisons to track progress, identify regression, even find flaky tests" employing that result tracking can help uncover issues that are harder to find [46]. The following are several result-tracking methods that can be used in Jest:

Snapshot testing is a feature in Jest that allows developers to capture the rendered output of functions or components and compare them to previous snapshots. If unexpected changes are found, Jest flags this as a failure, prompting developers to review the changes made. As discussed in Part 9.1: Test Automation Approaches of the paper, automated testing tools like Jest are emphasized in enabling rapid executions of repeatable tests. Furthermore, automating test data creation and comparing it with previously generated outputs aligns with the principles of snapshot testing, as mentioned earlier.

```
test('runs within acceptable performance limits', () : void => {
    const start : number = Date.now();
    someFunction();
    const end : number = Date.now();
    expect(end - start).toBeLessThan(100); // ensures function runs within 100ms
});
```

Figure 30. *Jest Test for Ensuring Function Runs Within Acceptable Performance Limits*

Jest's performance monitoring feature allows developers to analyze and track the performance of their code. This helps identify performance regressions over time, making sure that new changes do not impact the performance of the software negatively. Jest integrates with various tools like jest-performance and jest-stare for performance monitoring. These tools allow you to track how long tests take and compare their performance across different runs.

Detailed assertion logging helps track the behavior of tests by presenting detailed logs for each implemented assertion. Jest allows developers to view not only the test result but also the reason behind each assertion. This detail helps in debugging, particularly when dealing with complex cases where multiple assertions are involved. Jest's –verbose flag may be used to enable detailed logging of test results. This will output each individual test case with its assertions and results.

8 DEFECT PROCESSING

Defect processing involves identifying and handling material, product, or system flaws or issues. In software development, defects are often seen as a disconnect between a developer's intent and the program's behavior. According to Widder and Goues [48], bugs often stem from human error, resulting in defects in the code that can lead to system failures.

However, they note that not all defects cause immediate failures. Even minor defects can significantly impact cost, reputation, and user trust. These defects can arise at any stage of the software development process, from requirements engineering and design to implementation, testing, and maintenance [49]. Addressing defects earlier is crucial to minimizing adverse outcomes and ensuring that products meet quality standards and perform reliably. Companies can reduce waste, avoid costly mistakes, and improve performance by carefully processing defects.

Jest is a tool for better defect management that automates tests and improves code quality [50]. It allows developers to write unit and integration tests to detect issues earlier in the development phase before they become significant problems.

8.1 Defect Life Cycles

The Defect Life Cycle, or bug life cycle, tracks a software defect from identification to resolution. The process begins when a defect is reported and logged into a defect tracking system. It is then analyzed for cause and severity and may be assigned to a developer for fixing. After the defect is addressed, it is tested to ensure it functions correctly without introducing new issues. If successful, the defect is resolved; otherwise, it is reopened and worked on [51].

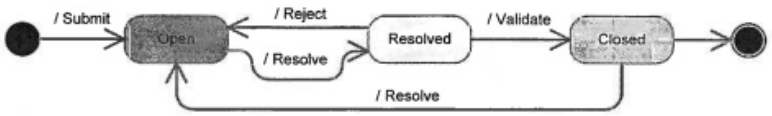


Figure 31. Three State Defect Model

According to Nindel-Edwards et al. [52], the cycle starts in the Submitted State, where defects are triaged, prioritized, and assigned for action. The triage team evaluates each defect's urgency and decides whether it should be addressed immediately, postponed, or closed. Defects may also be flagged for further review. Once a defect is approved for resolution, it moves to the Open State, where the developer works on the fix. The defect may be flagged for follow-up if delays occur due to resource or technical issues. If the fix is successful, the defect enters the Resolved State for verification, and if problems remain, it returns to the Open State for further work.

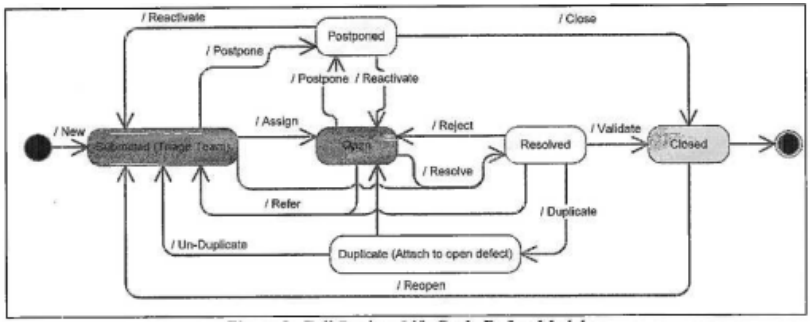


Figure 32. *Full Product Life Cycle Defect Model*
Note. This image is a Defect Model. From the "A Full Life Cycle Defect Process Model That Support Defect Tracking, Software Product Cycles, and Test Iterations (Communications of the IIMA) by Jim Nindel-Edwards, 2006"

Some defects may be classified as "Will not Fix," marked as duplicates or postponed. Defects marked as "Will not Fix" are considered low-priority or too complex to resolve in the current cycle, while duplicates are linked to a master defect. Postponed defects are

scheduled for future release cycles but must be monitored to avoid impacting overall product quality. If not managed well, postponed defects can accumulate and affect the final product [52].

In the requirements phase, defects are particularly impactful because they can affect the entire SDLC. "Wrong or missing requirements lead to a wrong or incomplete product, no matter how good the subsequent phases are" [53]. Detecting defects early in this phase helps prevent issues from propagating, saving time and resources in later stages. Effective defect detection is crucial for ensuring high-quality software.

To improve detection in the requirements phase, a new taxonomy has been proposed that correlates defects with their causes, filling gaps left by previous classifications. A comparison of defect detection techniques also highlights the effectiveness of various reading methods. This evaluation emphasizes the importance of selecting the best techniques for comprehensive defect detection [53].

Understanding the defect life cycle and the role of the requirements phase in identifying defects can significantly enhance software quality [54]. By detecting and addressing defects early, teams can reduce costly rework, improve efficiency, and ensure the software meets its desired standards.

In the initial stages, defects are triaged and prioritized. Through fast feedback loops, Jest helps teams quickly assess parts of the codebase most prone to errors. Developers can run tests frequently, ensuring new changes do not introduce additional defects.

Once a developer assigns a defect in the Open State, Jest facilitates fixing it with tools for mocking and simulating various scenarios. Jest's snapshot testing allows developers to test fixes separately, ensuring developers address issues without affecting other application parts [55]. After implementing the fixes, Jest can automatically run tests to verify that the defect has been resolved.

When a defect enters the Retest phase, Jest simplifies testing by allowing testers to quickly rerun existing test cases or create new ones targeting the fixed defect. If everything checks out, the defect can be closed; if not, it returns to the developer for further adjustments. Understanding these two concepts is critical for effective defect triage, which affect both the development process and the end-user experience.

8.2 Severity and Priority Defects

According to Hamilton (2024) [51] defect management is an essential part of development for ensuring a software product's quality and functionality. One critical aspect of defect management involves categorizing defects by their severity and priority. These two terms often need clarification but serve distinct purposes in guiding how defects are handled.

Severity refers to the impact a defect has on the product's functionality [56]. A defect can be critical, major, madorate, minor, or cosmetic, depending on how much it disrupts the system's operation. For instance, a critical defect might render a product completely unusable. In contrast, a cosmetic defect, such as a minor UI issue, might only affect the product's visual presentation without affecting its functionality. Severity is generally determined by the quality assurance (QA) engineer based on the technical aspect of the defect and its potential impact to impair product performance [57].

On the other hand, priority dictates the urgency with which a defect should be addressed. It reflects the

business impact, focusing on how soon a defect needs to be fixed in the project or release schedule context.

Priority is often influenced by customer requirements and is decided in consultation with project managers or stakeholders. For example, a high-priority defect could be one that hinders key functionalities that customers depend on, whereas a low-priority defect, like a minor UI inconsistency, can be resolved at a later stage [51].

The relationship between severity and priority can be complicated. A defect can be high in severity but low in priority, or vice versa. For instance, a defect in a flight reservation system, which prevents users from booking tickets (high severity), might not be addressed immediately if it occurs in a part of the system that is rarely used (low priority). Conversely, a seemingly minor issue, like a wrong logo on a website, might be of low severity but high priority, especially if it risks damaging the company's brand image[51].

Defect categorization by severity and priority plays a significant role in software reliability prediction. Serban and Vescan [57] highlight the importance of considering both factors when managing defects. Their research suggests that defects classified by severity and priority can provide valuable insights into software reliability. By understanding the nature of defects, teams can predict how these issues might affect the product over time. For instance, a high-priority defect, even with low severity, can accumulate over multiple releases, affecting overall reliability if not addressed promptly.

The defect triage process is essential for balancing the need for quick fixes with the long-term health of the software product. Triage involves evaluating defects based on their severity and priority to determine which should be addressed first. In this process, defects that are high in priority but low in severity might be fixed quickly to avoid immediate customer dissatisfaction, while high-severity defects are often prioritized for urgent resolution to prevent major system failures. The triage process helps allocate limited resources efficiently and ensures that the most critical issues are resolved in a timely manner [57].

In conclusion, categorizing defects by severity and priority is central to effective defect management. By understanding the difference between these two concepts and applying them thoughtfully, teams can improve not only the quality of the software but also its reliability over time. Predicting reliability through defect management practices, as proposed by [57], offers a powerful approach to managing software quality. Furthermore, the ability to triage defects based on severity and priority ensures that resources are allocated effectively and the product meets both customer and technical requirements. As software development continues to evolve, so will the strategies for managing defects, but the foundation of severity and priority remains critical to the process.

By using Jest’s capability to make automated tests, teams can catch high-severity defects early in the development cycle. This proactive approach ensures critical issues, which could severely impact functionality, are addressed promptly.

Jest also generates detailed coverage reports that highlight which parts of the code are tested. This visibility helps teams focus on high priority areas where defects are likely to occur, ensuring that critical functionalities are thoroughly vetted before release.

With Jest's mocking capabilities, developers can simulate various scenarios to test how defects might manifest under different conditions. This feature is particularly useful for assessing the severity of defects

by understanding their impact on user experience and system functionality.

Integrating Jest with Continuous Integration or Continuous Deployment (CI/CD) pipelines allows teams to prioritize defect resolution based on test results. High-priority defects identified during automated testing can be flagged for immediate attention, ensuring that resources are allocated efficiently.

Jest facilitates a rapid feedback loop between developers and testers. When defects are identified, they can be categorized by severity and priority based on their impact on the application. This classification helps teams make informed decisions about which defects to address first, balancing immediate fixes with long-term software health.

8.3 Writing Good Defect Reports

A defect report is a document that has concise details about what defects are identified, what action steps make the defects show up, and what are the expected results instead of the application showing error while taking particular step by step actions. [58]. Defect reports play a crucial role in communicating the gap between testers and developers, ensuring clarity about the issues found. Defects or bug reports provide crucial information to developers in fixing defects [59].

Defects can be fixed more quickly and accurately when the defect reports themselves are clearly/concisely written and useful to developers [59]. A well-written report eliminates guesswork, making it easier for developers to find the cause of the problem. This speeds up the debugging process and also reduces the chances of errors during the fix, making sure that the solution effectively resolves the defect.

The Eight Basic Elements of an Effective Defect Report [59] are essential for clearly communicating defects to the development team. First, the *Defect ID* is a unique identification number generated by a defect reporting or tracking tool whenever a tester raises a ticket. This helps in organizing and referencing defects easily. Next, the *Defect Description* provides a brief summary of the defect encountered, including details about the module and the source where the defect was found. It gives the team an immediate understanding of the problem area.

Following this, *Prerequisites* outline the steps and conditions required to recreate the defect. These prerequisites ensure that the defect can be consistently reproduced before further investigation. *Steps* then provide a clear, step-by-step guide that allows the development team to replicate the defect in their own testing environment, ensuring consistency in the reproduction process.

The *Actual Result* describes the behavior that occurs when the tester follows the steps, outlining exactly what happens when the defect is triggered. In contrast, the *Expected Result* defines the behavior that should have occurred according to the software's intended functionality. This helps the development team understand the discrepancy between the current behavior and the intended behavior.

Supporting evidence, such as a *Screenshot/Video Reference,* is also critical in a defect report. This can include screenshots, videos, logs, or crash reports, providing visual or textual proof of the defect occurrence. Finally, the *Testing Environment* section specifies the browser and operating system versions in which the defect was encountered. This information is crucial for the development team to recreate the issue in the same environment, ensuring accurate

troubleshooting and resolution. By including these eight elements, a defect report becomes a comprehensive tool for communication and problem-solving within the software development process.

Writing effective defect reports is essential for efficient communication between testers and developers.One of the key aspects of a good defect report is to *use a brief and concise title.* The title should clearly and succinctly describe the issue, allowing developers and testers to quickly understand the nature of the problem without needing to read through the entire report [60]. Along with the title, it's important to *write a clear and detailed description* of the defect. This should provide a brief explanation of the issue before detailing how to reproduce it. A clear description helps developers gain important context, enabling them to identify the problem more efficiently [60].

Another essential aspect is to *stay objective.* When writing a defect report, focus on reporting the facts of what you observed rather than making assumptions about the cause or potential solutions. This helps maintain clarity and prevents miscommunication [61]. Additionally, it is useful to *provide additional context* if necessary. If you notice any patterns or specific conditions under which the defect occurs, include these details to assist developers in diagnosing the issue [61].

Moreover, it is important to *be clear and concise* in all sections of the report. By providing information in a straightforward and precise manner, you minimize ambiguity and allow developers to quickly grasp the issue at hand [62]. Finally, it is crucial to *follow up and track* the defect's resolution. Keeping track of defects and ensuring they are addressed and resolved in a timely manner helps maintain project momentum and ensures that issues do not go unresolved [62]. By adhering to these guidelines, defect reports become more effective tools for identifying and resolving software issues.

8.4 Efficient Use of Defect Tracking Tools

Efficient use of defect tracking tools is crucial for improving software quality and streamlining the development process. Research has shown that proper implementation of these tools can significantly impact project outcomes and team productivity.

Choosing the right defect tracking tool is essential for efficient use. Factors to consider include user interface, usability, and specific features that align with team needs [63]. Popular tools like JIRA, Bugzilla, and Mantis offer various features suited for different team sizes and project complexities [64].

Integrating defect tracking tools into existing development workflows is crucial for maximizing their effectiveness. This integration ensures that bug reporting and resolution become seamless parts of the development process [65]. Implementing standardized reporting templates and consistent categorization of defects can significantly improve the efficiency of the tracking process. This approach facilitates easier triage, prioritization, and resolution of issues [66].

Utilizing collaboration features within defect tracking tools, such as commenting and assigning, can significantly improve team communication and bug resolution speed. Cloud-based solutions like Asana and Trello offer features that enhance team collaboration [65]. Conducting regular review meetings and maintaining a comprehensive defect log are essential practices for efficient defect tracking. These practices foster collaboration and provide valuable insights for future improvements [64].

Efficient use of defect tracking tools correlates with improved software quality. By catching problems early in the development process, these tools help save time and money while enhancing overall product quality [66]. To maintain efficiency, it's important to continuously evaluate and improve the defect tracking process. This includes monitoring key metrics such as defect density, mean time to detect (MTTD), and mean time to repair (MTTR) [66].

9 TEST AUTOMATION

In software development, automation testing refers to taking a repeatable manual process performed by a developer or tester and leveraging a test automation tool to automate the procedure[67]. It is a technique used to improve the execution speed or verification/checks or any other repeatable tasks in the software development lifecycle.

Automation helps accelerate software development by utilizing computers to do our dirty work. This approach dramatically benefits efficiency and speed because a computer can go through automated test scenarios in minutes or seconds [68]. Similarly with enhanced accuracy, it eliminates human error while going through test cases since the repetitive nature of manual testing often leads to mistakes and oversights.

Jest, A popular JavaScript testing framework, is leveraged for automation testing in a Javascript codebase. This framework comes bundled with exciting features such as snapshot testing, mocking, and a built-in testing tool for code coverage. One of its renowned features is zero configuration testing experience. This feature means that developers can now write tests for their project after installing Jest as a dependency in a codebase with no adjustments. Also, developers typically use Jest to automate unit, integration, and end-to-end testing.

9.1 Test Automation Approaches

There are many determinants of the selection of test automation approaches, such as the application's nature and complexity, defined goals and parameters for the testing procedure, available tools and technologies, and sometimes time or budgetary constraints and workforce limitations [69]. Thus, this customization to align with the factors makes the testing process more productive and structured and ensures that all functional and non-functional criteria are satisfactorily met.

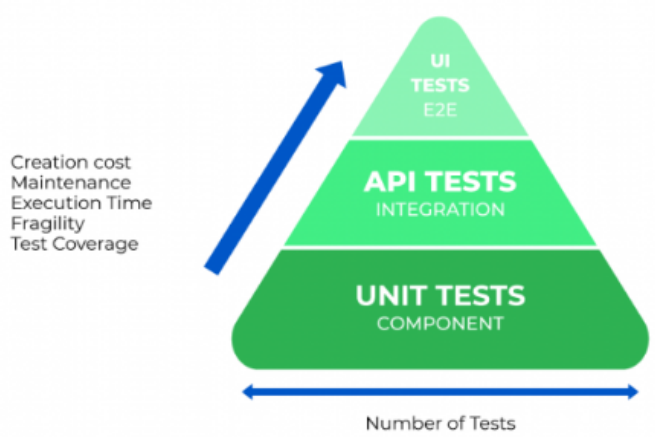


Figure 33. *Test Automation Pyramid*
Note. This image is about the test automation pyramid. From the "Succeeding with Agile: Software Development Using Scrum" by Mike Cohn, 2009.

Figure 31 (Cohn) illustrates that costs to support, deliver, and pay for tests increase exponentially through

the pyramid. This results in the best way – putting tests as low in the pyramid as possible to maximize ROI.

In contrast, UI tests are brittle, time-consuming, and costly. The other layers of Unit and API tests take precedence over these because they offer better test coverage

with low overhead in the creation, execution, and maintenance of tests [70]. However, it is essential to acknowledge that the test automation pyramid should not be a one-size-fits-all solution. Collaboration with the team must be made to identify the most critical risks and determine how automation can counter them successfully.

Let's understand some of the key approaches with examples using Jest.

1. *UI automation*

This approach involves automating user interface interactions to validate the application's behavior from the end user's perspective [69].

Figure 34. *ui.test.js*

For UI automation, Jest can then be used in conjunction with the React Testing Library to effectively simulate user interactions and verify the correctness of how the UI works, simulating actions a user might take, like clicking a button and checking for the UI changes that would happen consequentially, like the label on a button to ensure expected functionality.

2. *API automation*

In API automation, tests interact directly with the application's backend through its APIs (Application Programming Interfaces) to validate functionality, data, and integrations [69]

Figure 35. *api.test.js*

In the context of API automation, one could use a more advanced feature provided by Jest for mocking or might instead use a library like Axios or Fetch. It involves simulating an API request and checking the function's interaction with the backend for proper communication and implementation.

3. *Unit Test automation*

This approach focuses on testing individual units or components of the application in isolation, typically at the code level [69]

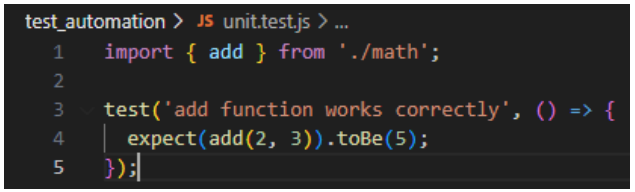


Figure 36. *unit.test.js*

Jest proves brilliant in unit testing by focusing on verifying isolated discrete functions or components. The following figure checks the actual operation of an independent function, so it is sure of its correctness at the code level.

4. *Performance testing automation*

Performance testing involves assessing how an application performs under various load conditions [69]. Automation helps simulate thousands of virtual users to measure performance metrics.

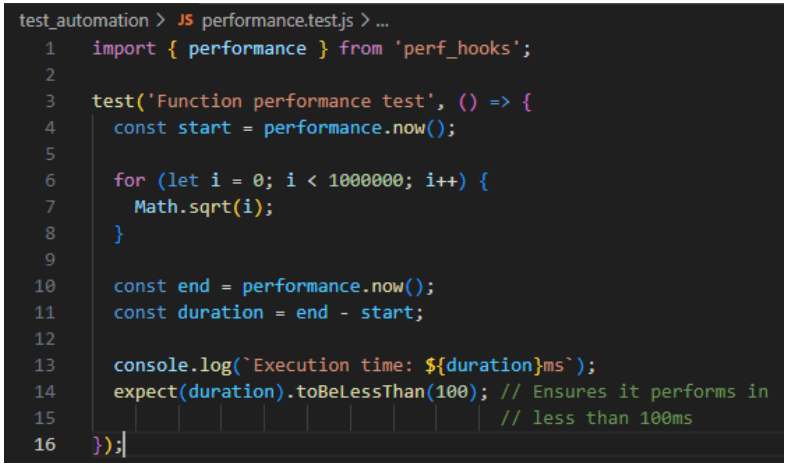


Figure 37. *performance.test.js*

Although Jest is not explicitly intended as a performance testing framework, it can still be used quite effectively in combination with libraries like benchmark.js or Node.js performance hooks to assess and investigate the computational process efficiency.

Overall, UI automation verifies that the user interface does the right job by ensuring an uninterrupted experience for the user. Similarly, API automation verifies the effective working of backend interactions, thereby checking application logic and data transactions. Unit test automation focuses on isolating and ensuring the accuracy and reliability of independent components or functions on the code level. Performance testing automation checks the responsiveness and stability of the system against various load conditions. Tests may be configured to provide an exhaustive report like jest-html-reporter. Tests report helpful information to the development team to maintain and improve quality software.

Conversely, test automation methodologies have organized frameworks and strategies aimed at optimizing automated testing in software development. The primary goals of such methodologies are to optimize operational efficiency while ensuring uniformity [69]. These various methodologies also address multiple aspects of a testing activity-from planning through execution to maintenance. Some common methodologies include the following:

Behavior-driven development (BDD) refers to a methodology that uses practical instances to guide the software development process. In this respect, it ensures cooperation between developers, testers, and business analysts in their collective understanding of requirements[70]. BDD focuses on writing tests before producing actual code. Initial steps involve acceptance tests, testing from a user's view on behavior in the software, and then checking single components with unit tests. It bridges many communication gaps and ensures the software meets business objectives adequately.

In the context of Jest, it is primarily a framework for unit testing, and it is possible to use it in combination with libraries such as Jasmine or Cucumber to make it also support BDD. The mentioned libraries enable tests to be written more along the lines of natural language, in accordance with the BDD approach. Although Jest itself does not fully support BDD, using it with the correct tools can lead to this approach.

Keyword-driven testing is a methodology that uses keywords to represent the actions or operations performed in the application under test [69]. Test scripts are then produced by combining reusable keywords that allow people to construct and maintain tests without requiring in-depth programming knowledge . The abstraction of test actions into simple, easy-to-understand vocabulary makes test construction feasible even for people with less technical expertise in a way that maintains efficiency and reusability throughout the test execution process.

In the scope of Jest testing, Jest does not support keyword-driven testing as a methodology in its own right; however, it is easily adapted to work toward this end by creating reusable functions or bespoke test helpers. This approach is somewhat abstracted, allowing testers to specify test procedures in less technical terms irrespective of their level of programming experience. However, the framework must be established by hand to mimic a keyword-driven approach.

Test-driven development is an engineering practice that follows a "Test-First" approach by which the developers write unit tests before writing the functional code. In TDD, developers verify that a test exists before implementing a function. The test fails at the outset because the corresponding function has not been designed, and it drives the creation of the function by passing the test [70]. It encourages continuous testing and ensures the code is appropriately tested from an early stage to attain the enhanced code quality and reliability of the software.

In Jest's context, Jest is a natural fit for Test-Driven Development. The "test-first" philosophy is at the heart of Jest's design; developers can write tests before anything else. With Jest, developers write unit tests first, and as they develop the application, they can repeatedly run those tests to ensure the code behaves as expected. Jest's fast feedback cycle makes it perfect for TDD.

Data-driven testing is a methodology that separates test data from the test scripts. With the help of data-driven testing, the same test case is executed

multiple times with different input data sets, which validates various scenarios and also increases test coverage by efficiently testing numerous data permutations. It works to streamline the testing process, ensuring that different conditions are met without duplicating test scripts [69].

From Jest's perspective, Jest does not natively support this methodology. Although parameterized tests could be implemented, suppose using the test.each() or iterating over different data sets and thus emulating data-driven testing within Jest. In that case, the same test logic can be run against many different data sets to increase test coverage very efficiently.

Continuous Testing is a methodology that involves automatic testing throughout the entire software delivery pipeline, from development to deployment. This method ensures that code changes are consistently checked against a wide range of tests, allowing for rapid feedback, thus quickly pinpointing defects [69]. It helps maintain quality during the developing phase, reducing the chances of problems surfacing during deployment.

Within Jest's framework, Jest can be integrated into a continuous integration (CI) pipeline to support continuous testing. By running Jest tests automatically upon each modification in the code, one can ensure that the code validates constantly throughout the software delivery lifecycle. At the same time, Jest's fast execution speed , and ability to interact with CI tools such as Jenkins or Travis CI, make it an excellent choice for continuous testing in agile development environments.

In conclusion, Jest can be easily adopted for test-driven development and continuous testing but is flexible enough to adapt for behavior-driven development, keyword-driven testing, or even data-driven testing using a few extra libraries or even custom configurations. Considering these factors, the uniform acceptance of such a framework makes it ideal for developers who want to facilitate robust, comprehensive automated testing strategies.

9.2 Automation Process

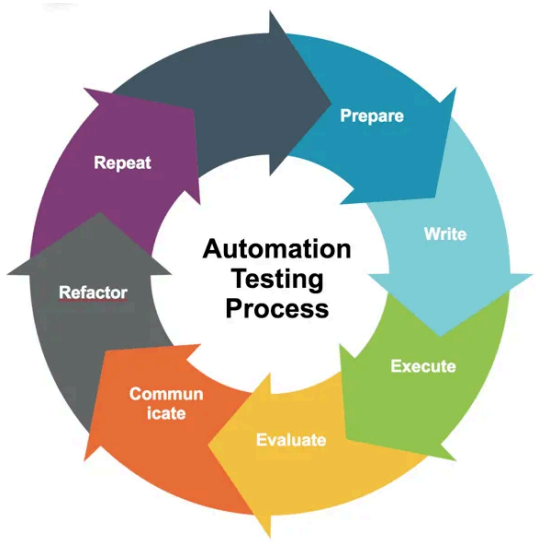


Figure 38. *Automation Testing Process*
Note. This image is about the automation testing process. From the "What is Automation Testing (The Ultimate Guide 2024)" by Joe Colantionio, 2024.

As shown in Figure 38, the testing cycle illustrates how each phase builds upon the previous one to ensure thorough, repeatable, and reliable automated testing.

The following six steps outline an effective automation process for achieving reliable automated software testing, particularly when utilizing Jest for JavaScript applications.

1. *Preparation*

an automation testing strategy that is only as good as the preparation behind it [71]. Functional testing goals need to be clear before test writing with Jest. It entails choosing what specific features or parts to test, collecting the relevant data needed for the tests, and deciding on the expected results. It is where the cooperation of the complete testing team is essential so that everybody knows what to do and there is no mismatch or misalignment. In the context of Jest, it involves setting up the framework, Installing dependencies, and configuring the environment (e.g., jest.config.js) to suit the project's requirements.

2. *Writing Tests*

The next critical step would be transforming the requirements into actionable test cases. With Jest, developers can write modular and isolated tests for each component or functionality. Tests need to show where one expects to start and where one expects to end [81]. Each test must use suitable assertion methods (such as toBe, toMatchObject, and toThrow) to ensure the application behaves correctly. Jest promotes writing independent tests. It facilitates more excellent reusability and fewer dependencies. For example, a snapshot test can check the rendering output of a React component to ensure it matches the specifications of the design.

3. *Execution*

Tests must be executed and repeatable, yielding meaningful results. Running automated testing scripts with Jest can be achieved through simple commands, like (npx jest), in an identical and efficient process. Executing each test multiple times is essential to verify stability and identify potential issues with flaky tests [82]. Additionally, Jest's watch mode can be used (just --watch) to automatically run the same tests after modifying the code for immediate developer feedback.

4. *Evaluation*

The results of tests need to be assessed to ensure the automation process is successful [83]. Jest provides a rich test report containing information about the tests that have passed, failed, or been skipped. In some cases, manual verification may be performed to check whether the automated tests are accurate. Additionally, if the tests fail to verify the expected behavior, they should be updated to include all the necessary assertions. Built-in debugging tools in Jest, like enhanced error messages, make the problem-identification process much easier during this phase.

5. *Communication*

Clear communication among the testing team members is foundational for maintaining transparency and accountability [73]. Jest allows the reporting of test results by integrating with various reporting tools, including jest-html-reporter and CI/CD pipelines. Test results, including flaky or persistently failing tests, must be communicated as soon as possible to ensure that correct actions are taken. Maintaining a reliable test suite brings confidence to the testing team in the results while reducing the risk of missing test failures.

6. *Refactor/Repeat*

The underlying philosophy of automation testing is continuous improvement. Where the tests are unstable or inconsistent, the test code must be refactored to stabilize tests once again [74]. Jest allows modular design and the application of helper functions that make refactoring less complicated. Where a test is redundant or outdated and cannot be corrected in time, it should be removed to maintain the efficiency of the test suite. The test suite is always robust and aligned to project requirements in this iterative process.

In conclusion, These processes allow teams to develop a streamlined, efficient test automation approach that follows the six testing process steps. Such modular structure, easy API, and detailed reporting capabilities in Jest make it a massive tool for modern test automation. Following best practices makes test automation maintainable, reliable, and a non-optional constituent of the development cycle.

9.3 Unit Test Frameworks

Unit testing is the first critical stage of software testing, focused on the individual parts or units of an application that each works properly[74]. It is done during the development stage, which involves breaking up an application's code into small pieces and evaluating each piece of fragmented solitude. Then, the earlier bugs will be detected to minimize the possibility of costly errors elsewhere in the development cycle. This process typically begins by creating diverse test cases to ascertain the application's components. These test cases are validated and rewritten if necessary for correctness. The code is next inspected for adherence to known standards for coding before test cases are run as a final step in ensuring that each unit may function correctly in execution.

Regrettably, a number of developers tend to neglect unit testing, frequently resulting in a rise in software defects, elevated expenses associated with debugging, and additional challenges during subsequent phases of development. The implementation of automated unit testing through frameworks not only facilitates the process but also guarantees the accuracy and dependability of the code.

This is where unit testing frameworks come into play, providing an organized approach towards ensuring the functionality of different code pieces is correct. These frameworks are vital when maintaining quality code, ensuring reliability in delivered software, and debugging [72]. Tools and libraries for unit testing are carefully designed to help the developers write, manage, and efficiently execute unit tests. They build up a controlled environment where developers can methodically analyze every aspect of the codebase to ensure that the functionality meets expectations, thereby bringing development and quality assurance together to ensure durable and reliable software.

Jest — an open-source testing framework built in JavaScript, primarily designed for use with React and React Native applications. Front-end unit tests often require extensive and time-consuming configuration, making them less practical. Jest addresses this issue, by streamlining the process and reducing the configuration burden significantly. Besides being applicable in React-centric applications, Jest is capable enough to evaluate a wide range of JavaScript functionalities, particularly related to the rendering of web applications within browsers. Its prevalent use for the automation of tests in browsers has made it one of the most prominent testing frameworks for JavaScript.

Jest comes with an all-inclusive suite of an assertion library, test runner, and an integrated mocking library all together in a singular tool. The nature of Jest is simple, and that makes it a prime case in the testing of JavaScript libraries and frameworks-including AngularJS, Vue.js, Node.js, Babel and TypeScript. Such flexibility and ease of use had set Jest as the favorite framework for developers performing unit testing of JavaScript [75].

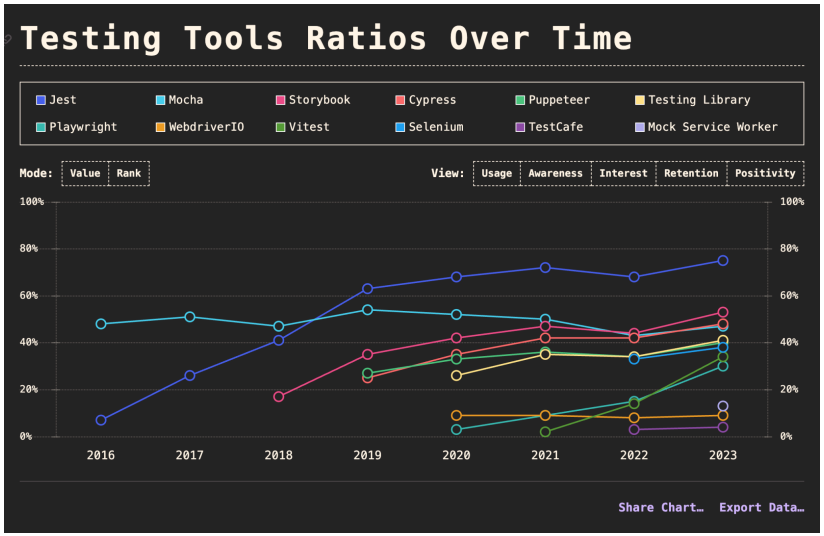


Figure 39. *Javascript testing tool awareness*
Note. This image is about the popularity of different Javascript testing tools. From "State of JavaScript 2023", 2023.

As shown in Figure 39 (State of Javascript 2023), Jest has steadily increased in recognition and usage through the years; Jest has become one of the leading unit testing frameworks. The evidence reflects that the awareness rate achieved by Jest reached 95% in 2023, indicating an extreme level of acknowledgment by the developer community. This kind of widespread recognition is probably one of the reasons for its increased popularity, as developers are becoming increasingly aware of its functionalities and characteristics.

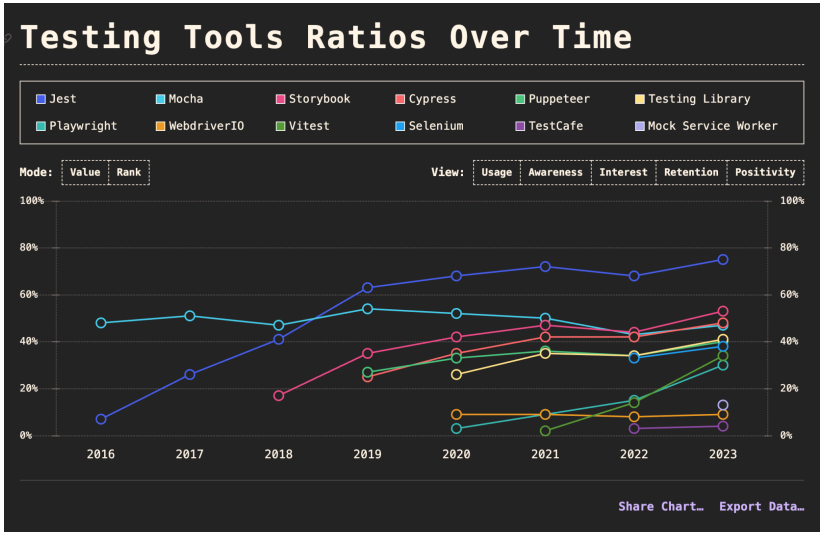


Figure 40. *Javascript testing tool usage*
Note. This image is about the popularity of different Javascript testing tools. From "State of JavaScript 2023", 2023.

The continued enthusiasm for Jest is notable, as evidenced by a 67% interest rate in 2023. Although this represents a modest decline from the high of 81% observed in 2019, it nonetheless indicates a substantial level of curiosity and investigation regarding Jest among developers. This persistent interest implies that developers are not merely cognizant of Jest but are

actively utilizing it and investigating its capabilities for testing applications built on JavaScript.

The most significant metric of Jest's adoption is its steady upward trend in usage. While only 7% of developers adopted the tool in 2016, by 2023, the percentage increased to 74%, which shows a tremendous growth in the actual number of users working with the framework. The upward trend reflects that developers are increasingly confident with Jest, which they see as effective and reliable for testing purposes.

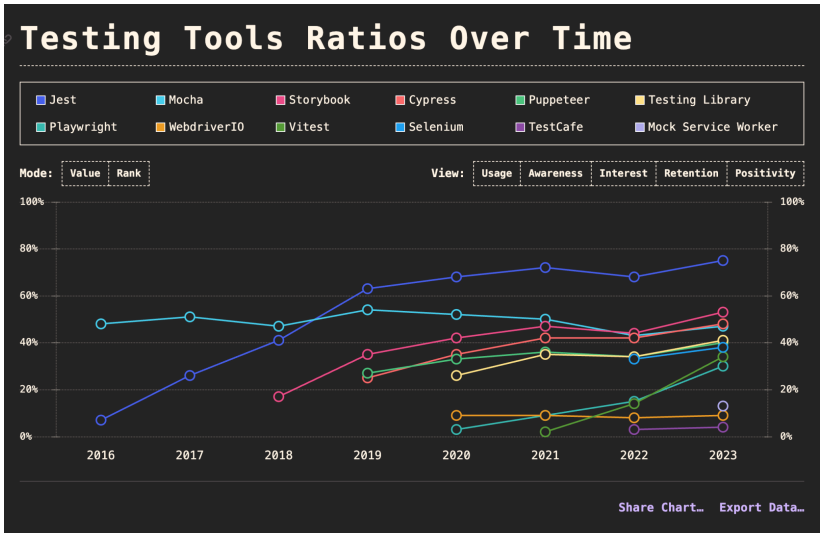


Figure 41. *Javascript testing tool retention*
Note. This image is about the popularity of different Javascript testing tools. From "State of JavaScript 2023", 2023.

In addition, as Jest increases in use, it exhibits a very high retention rate, which has stayed above 79% for several years. This trend suggests that developers using Jest are likely to continue using the technology. Also, the positivity rate of 73% achieved by the framework in 2023 shows a high level of user satisfaction, which usually indicates that Jest users are satisfied with its performance and are likely to recommend it to other developers.

Its built-in solution for JavaScript testing is a significant reason for the widespread acceptance of Jest. By including an assertion library, a test runner, and a native mocking library, Jest provides a suite of tools for developers to fall back on while developing applications. This simplicity and ease make it very appealing to developers who work with JavaScript and React applications, where it helps avoid the complexities often associated with front-end testing.

In conclusion, this maturity can be attributed to the endorsement by the development team behind React and its usage and support by Facebook. For a programmer with several years of experience, it is imperative to have a framework that has a strong community and active assistance (Taleb, 2024). Jest delivers in this respect, assuring consistent updates and improvements and helping with problem-solving. The stamp of approval by a tech giant like Meta comes with an impressive amount of confidence and reliability. This means that if anyone runs into any issues, the probability of finding a solution relatively quickly increases through comprehensive documentation and a sizable, interested community of developers who might be able to help debug within hours if not minutes. The extent to which support is provided makes Jest appealing and reliable for application developers.

9.4 Automation of Regression Testing

Regression Testing is a software testing technique that ensures the changes in the program, including changed code, have not affected the good features and functionality built into the application. It is done by running all or some of the previously executed test cases again to ensure the software is not malfunctioning after applying new changes. Regression testing aims to see whether any newly introduced changes in the code-bugs, rectifications, feature improvement, or performance are causing some unanticipated impacts that may further interfere with the system's smooth operation. Essentially, it ensures that the code that ran perfectly earlier in the system continues to do so as correctly after the emergence of changes [77].

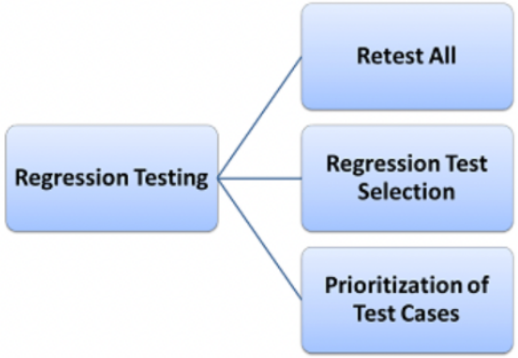
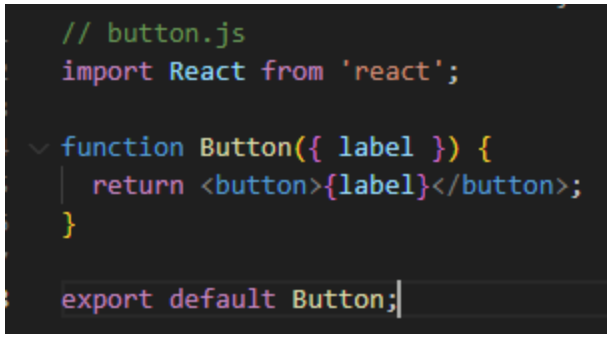


Figure 42. *Regression Testing Note. This image is about the regression testing model. From "Regression Testing" by Jorge Casariego, 2020.*

Automated regression testing, on the other hand, is the methodology in software testing where software testing uses automated tooling to ascertain that the codebase changes, including the addition of new features or the correction of defects, have not adversely affected the pre-existing functionality of the software [78]. This process ensures that the provided software will operate as expected after the impact has been incurred.

Automated regression testing is perhaps one of the most essential methodologies in software testing. It involves the use of automation tools in order to execute a predetermined set of regression tests. The intent behind this approach is to check whether the changes done in the code have not spoiled the pre-existing working functionality of any software application. One of the most accurate and efficient ways of validating software performance, automated regression testing allows for consistent, periodic testing [79]. It serves as a safeguard, quickly catching and preventing potential regressions, thereby avoiding the propagation of defects to the production environments.

Jest is also a critical tool in automated regression testing because it offers a highly efficient and reliable test-case-execution framework that ensures new code cannot break older functionality. It is an automated testing tool that enables developers to write tests covering all the application aspects, from simple unit tests to complex integrations. Such tests automatically run the moment the code is committed with any change, thus providing quick feedback on whether such new changes have any regressions. Here's a simple test to show how regression tests on Jest work:



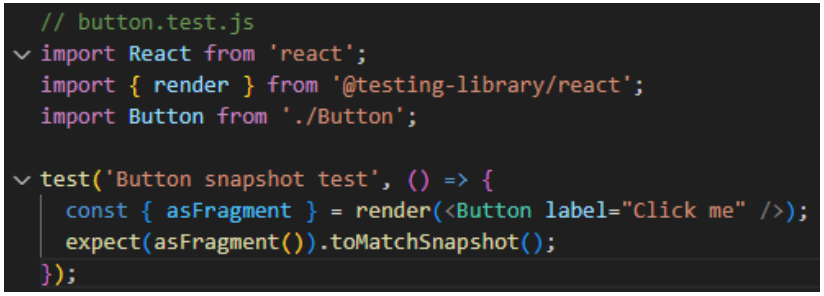
```
// button.js
import React from 'react';

function Button({ label }) {
  return <button>{label}</button>;
}

export default Button;
```

Figure 43. *button.js*

This figure shows the Button component in a simple React application. It takes a label prop and renders a button element with the text of the label. In this component, the Button takes a label prop and renders a <button> HTML element with the label as its content. It's a straightforward component used for demonstrating snapshot testing.

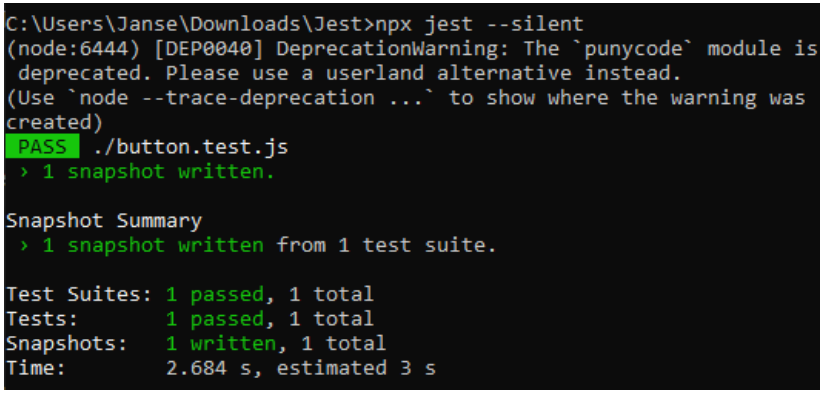


```
// button.test.js
import React from 'react';
import { render } from '@testing-library/react';
import Button from './Button';

test('Button snapshot test', () => {
  const { asFragment } = render(<Button label="Click me" />);
  expect(asFragment()).toMatchSnapshot();
});
```

Figure 44. *button.test.js*

This figure demonstrates how Jest is used to perform snapshot testing on the Button component. The test ensures that the rendered output of the component remains consistent across future changes. Here, @testing-library/react is used to render the Button component with the label "Click me". The asFragment() function from React Testing Library captures the rendered HTML as a document fragment, and toMatchSnapshot() is a Jest matcher that compares the current rendered output to a previously saved snapshot.



```
C:\Users\Janse\Downloads\Jest>npx jest --silent
(node:6444) [DEP0040] DeprecationWarning: The `punycode` module is
 deprecated. Please use a userland alternative instead.
(Use `node --trace-deprecation ...` to show where the warning was
created)
 PASS  ./button.test.js
 › 1 snapshot written.

Snapshot Summary
 › 1 snapshot written from 1 test suite.

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 written, 1 total
Time:        2.684 s, estimated 3 s
```

Figure 45. *First Run*

As shown in Figure 45, Jest will generate a snapshot file (typically under a __snapshots__ directory), storing the current rendered output of the component.

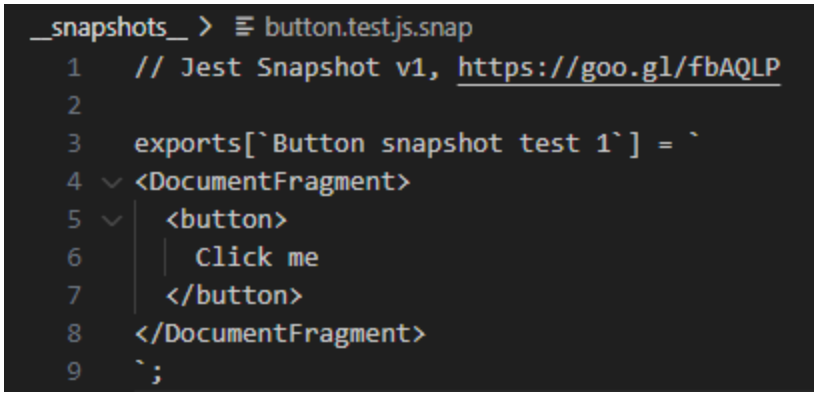


Figure 46. *button.test.js.snap*

On future test runs, Jest compares the newly rendered output to the stored snapshot. If any changes occur, Jest highlights the differences, indicating a regression in the component's output.

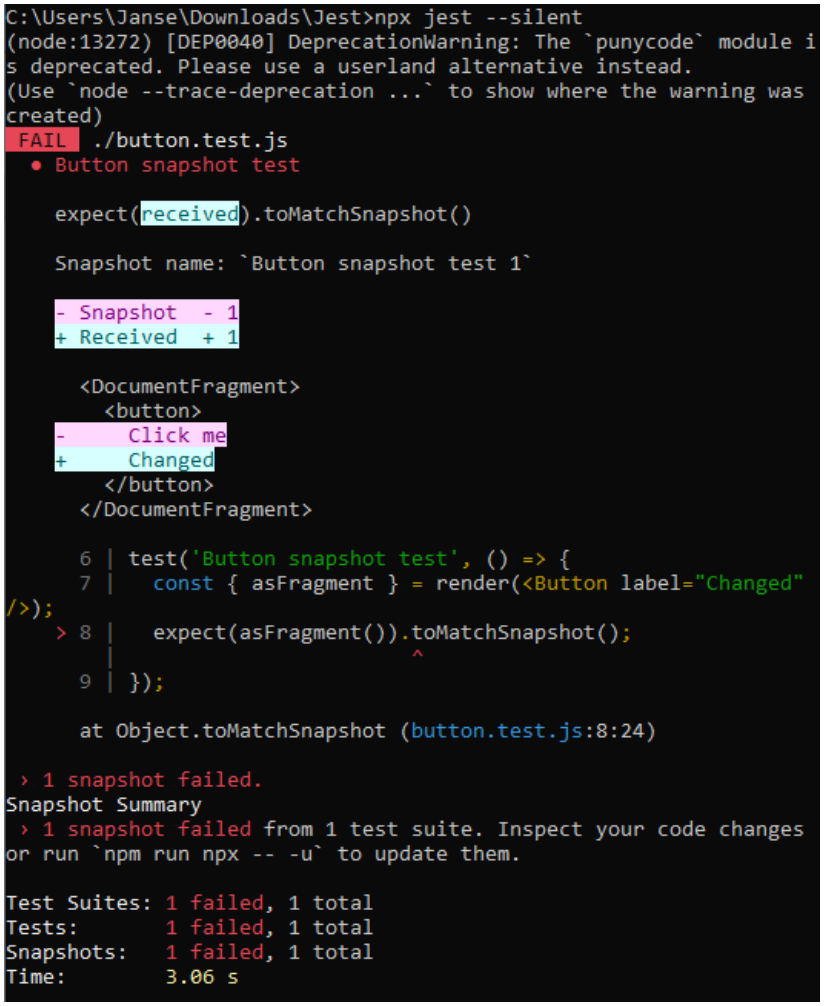


Figure 47. *Subsequent Runs*

This approach of using Jest's snapshot testing for regression testing is simple yet powerful to ensure stability in software. Snapshot testing automatically

tracks the output of components or functions and notifies developers immediately when there have been unintended changes to that output, thus protecting the integrity of the evolving application.

In line with this, Jest also supports seamless integration with tools like Jenkins, GitHub Actions, or GitLab CI that automatically lets test execution occur with each build. Whenever any one of the developers checks in their new code, it automatically triggers the CI pipeline to make Jest execute regression tests without any user intervention. This meant immediate feedback if changes to the code being pushed introduced regressions or the existing functionality was broken. Upon errors flagged, the build would fail, and the team would be alerted for correction before hitting the production stage.

In conclusion, Jest allows teams to maintain high-quality Javascript software while accelerating development. In current rapidly devolving environments, automated regression testing is necessary to ensure the efficiency and reliability of the application. The repetitive nature of regression tests makes automation particularly useful, as it minimizes human error, maximizes test coverage, and reduces the execution time of a test [79]. Through continuous validation of new and existing functionalities introduced into the system, regression testing automation fosters the development of robust, high-performance applications with minimal risks of code change.

9.5 Minimizing Test Automation Maintenance

Minimizing test automation maintenance is crucial for ensuring automated testing processes' sustained efficiency and effectiveness [80].
Embracing an automated test suite in software development offers significant advantages in today's fast-paced environment. Test automation allows organizations to streamline workflows, enabling the rapid creation of high-quality software. It helps teams detect regressions quickly, facilitating prompt bug fixes and freeing developers to focus on more impactful work . Furthermore, it provides the confidence to release new features that frequently enhance the customer experience. However, these benefits rely heavily on consistent and diligent test suite maintenance.

Maintenance for test automation is essential for long-term success. As the application evolves with codebase changes and system updates, tests must be carefully updated to reflect the new conditions. It involves writing or modifying new scripts to validate features, prevent recurring bugs, and maintain a smooth development process [78]. Ignoring this critical task can help progress and maintain present and future performance.

The following are some of the best practices that outline how to minimize test automation maintenance with Jest.

Centralizing configurations, locators, and reusable data simplifies test maintenance. In Jest, these can be implemented through shared files, constants, or helper functions. Centralized updates reduce the time required for maintenance and ensure consistency across the test suite.

```
C:\Users\Janse\Downloads\Jest>type jest-structure.txt
project-root/
+-- src/                        # Application source code
¦   +-- sum.js
¦   +-- otherModules.js
+-- tests/                      # Test files
¦   +-- unit/                   # Unit test files
¦   ¦   +-- sum.test.js
¦   ¦   +-- otherModules.test.js
¦   +-- integration/            # Integration test files
¦   +-- helpers/                # Reusable utilities for tests
¦       +-- testUtils.js
¦       +-- testData.js
+-- config/                     # Centralized configurations
¦   +-- jest.config.js          # Jest configuration
¦   +-- locators.js             # Shared locators or test constants
+-- node_modules/               # Dependencies
+-- package.json                # Project dependencies and scripts
+-- README.md                   # Documentation
```

Figure 48. *Jest Structure*

Writing clean and modular test codes reduces test maintenance by making test cases legible and reusable; developers can minimize the intricacy of updating and modifying tests. Jest's snapshot testing feature supports this process by automatically updating test cases to reflect UI changes, thus decreasing manual intervention. Similarly, employing variables and helper functions instead of hard-coded values further enhances code reusability and maintainability.

```
C:\Users\Janse\Downloads\Jest>npx jest --updateSnapshot
 PASS  ./button.test.js
  › 1 snapshot written.
 PASS  ./sum.test.js

Snapshot Summary
  › 1 snapshot written from 1 test suite.

Test Suites: 2 passed, 2 total
Tests:       3 passed, 3 total
Snapshots:   1 written, 1 total
Time:        0.852 s, estimated 1 s
Ran all test suites.
```

Figure 49. *Jest Snapshot*

Automation of test reports allows a team to monitor and track their results efficiently. Fortunately, Jest supports integration with reporting frameworks, such as jest-html-reporter. It enables detailed insights into test coverage and performance. These reports facilitate data-driven decision-making and continuous improvement in the automation process.

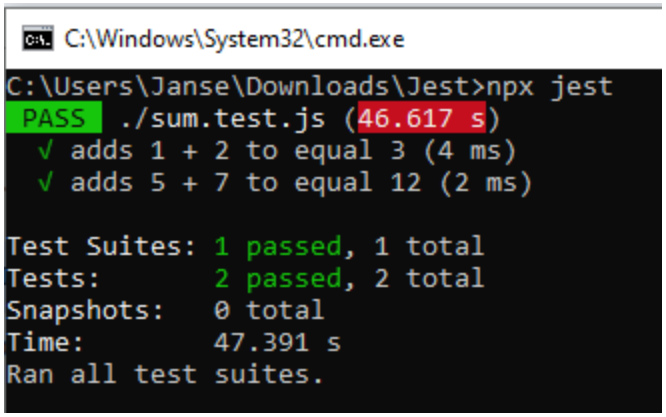


Figure 50. *Jest's Terminal results*

***Figure* 51.** *Jest-html-reporter*

Integrating continuous testing through continuous integration (CI) pipelines is crucial for maintaining resilient Jest test suites. CI platforms, such as GitHub Action or Jenkins, facilitate frequent test execution, enabling developers to identify and resolve issues early. This active approach minimizes extensive maintenance and promotes a stable development environment.

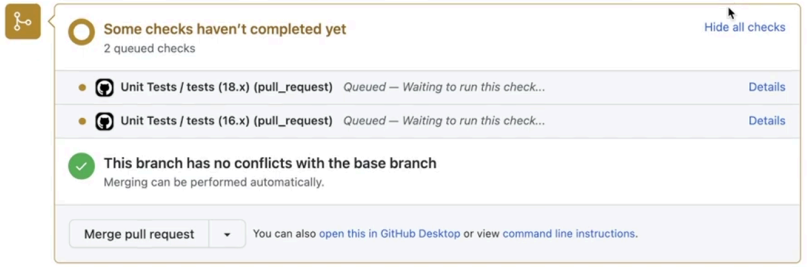


Figure 52. *Jest integrated with Github Actions*

By employing these best practices, Jest users can optimize the maintainability of their automated test suites; overall, it enhances their effectiveness in sustaining agile development workflows and delivering high-quality software.

APPENDIX A: LIST OF REFERENCES

[1] Desikan, S., & Ramesh, G. (2006). Software testing: Principles and practices (6th impression). Dorling Kindersley. https://books.google.com.ph/books?id=Yt2yRW6du9wC&newbks=0&printsec=frontcover&hl=en&source=newbks_fb&redir_esc=y#v=onepage&q&f=false

[2] GeeksforGeeks. (2022, August 30). Difference between Unit Testing and Integration Testing. GeeksforGeeks.

[3] GeeksforGeeks. (2024, August 26). Difference between system testing and acceptance testing.

[4] Lehtinen, T. O. A., Mäntylä, M. V., Vanhanen, J., Lassenius, C., & Itkonen, J. (2014). Perceived causes of software project failures: An analysis of their relationships. Information and Software Technology, 56(6), 623–643. https://doi.org/10.1016/j.infsof.2014.01.015

[5] Verner, J., Sampson, J., & Cerpa, N. (2008). What factors lead to software project failure? 2008 Second International Conference on Research Challenges in Information Science, 11–18. https://doi.org/10.1109/rcis.2008.4632095

[6] Hsueh, S., & Ranasaria, A. (2008). A study of exception-handling errors in large software systems. Empirical Software Engineering, 13(4), 385–414.

[7] The Jest testing framework: our top five features. (n.d.). https://www.adesso.de/en/news/blog/the-jest-testing-framework-our-top-five-features-2.jsp

[8] Jest architecture. (n.d.). [Video]. https://www.lambdatest.com/jest

[9] Shahrokni, A., & Feldt, R. (2013). A systematic review of software robustness. Inf. Softw. Technol., 55, 1-17.

[10] Garcia, A. F., Rubira, C. M., Romanovsky, A., & Xu, J. (2001). A comparative study of exception handling mechanisms for building dependable object-oriented software. Journal of systems and software, 59(2), 197-222.

[11] Cacho, N., Barbosa, E. A., Araujo, J., Pranto, F., Garcia, A., Cesar, T., ... & Garcia, I. (2014, September). How does exception handling behavior evolve? an exploratory study in java and c# applications. In 2014 IEEE International Conference on Software Maintenance and Evolution (pp. 31-40). IEEE.

[12] Cacho, N., César, T., Filipe, T., Soares, E., Cassio, A., Souza, R., Garcia, I., Barbosa, E. A., & Garcia, A. (2014). Trading robustness for maintainability: an empirical study of evolving c# programs. Proceedings of the 44th International Conference on Software Engineering.

[13] Osman, H., Chis, A., Corrodi, C., Ghafari, M., & Nierstrasz, O. (2017, May). Exception evolution in long-lived Java systems. In 2017 IEEE/ACM 14th

International Conference on Mining Software Repositories (MSR) (pp. 302-311). IEEE.

[14] Filho, J. L. M., Rocha, L., Andrade, R., & Britto, R. (2017). Preventing erosion in exception handling design using Static-Architecture conformance checking. In Lecture notes in computer science (pp. 67–83). https://doi.org/10.1007/978-3-319-65831-5_5

[15] Ebert, F., Castor, F., & Serebrenik, A. (2015). An exploratory study on exception handling bugs in Java programs. Journal of Systems and Software, 106, 82-101.

[16] Sawadpong, P., & Allen, E. B. (2016, January). Software defect prediction using exception handling call graphs: A case study. In 2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE) (pp. 55-62). IEEE.

[17] de Pádua, G. B., & Shang, W. (2017, May). Studying the prevalence of exception handling anti-patterns. In 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC) (pp. 328-331). IEEE.

[18] de Pádua, G. B., & Shang, W. (2018, May). Studying the relationship between exception handling practices and post-release defects. In Proceedings of the 15th International Conference on Mining Software Repositories (pp. 564-575).

[19] Sinha, S., & Harrold, M. J. (2000). Analysis and testing of programs with exception handling constructs. IEEE Transactions on Software Engineering, 26(9), 849-871.

[20] Martins, A. L., Hanazumi, S., & De Melo, A. C. (2014, July). Testing java exceptions: An instrumentation technique. In 2014 IEEE 38th international computer software and applications conference workshops (pp. 626-631). IEEE.

[21] Roger, S. (2024, November 20). Exception / error handling in software testing. H2kinfosys Blog. https://www.h2kinfosys.com/blog/exception-or-error-handling-in-software-testing/

[22] Das, S. (2024, October 7). What is System Testing? (Examples, Use Cases, Types) | BrowserStack. BrowserStack. https://www.browserstack.com/guide/what-is-system-testing

[23] GeeksforGeeks. (2024, August 26). Difference between System Testing and Acceptance Testing. GeeksforGeeks. https://www.geeksforgeeks.org/difference-between-system-testing-and-acceptance-testing/

[24] System Testing vs Acceptance Testing - javatpoint. (n.d.). www.javatpoint.com. https://www.javatpoint.com/system-testing-vs-acceptance-testing

[25] Budisaputro, C., Anardani, S., Riyanto, S., & Kusdwiadji, A. (2024). Medical Record Information system testing using user acceptance testing to determine system quality. Brilliance

Research of Artificial Intelligence, 4(1), 422–426. https://doi.org/10.47709/brilliance.v4i1.4451

[26] Cimperman, R. (2006). Uat defined: A guide to practical user acceptance testing (digital short cut). Pearson Education.

[27] Hoque, S. (2018). Full-stack react projects: Modern web development using react 16, node, express, and mongodb. Packt Publishing Ltd.

[28] Myers, G. J., Elmendorf, W. R., & IEEE-ANSI. (1979, 1973, 1989). The art of software testing; Cause-effect graphs in functional testing (TR-00.2487); IEEE guide for the user of IEEE standard dictionary of measures to produce reliable software (IEEE Std 982.2-1988). Wiley; IBM Systems Development Division; IEEE Standards Board and ANSI

[29] Kaner, C. (2003). On scenario testing. Software Testing and Quality Eng. Magazine, 16-22.

[30] Wichansky, A. M. (2000). Usability testing in 2000 and beyond. Ergonomics, 43(7), 998–1006. https://doi.org/10.1080/001401300409170

[31] What is Usability Testing? (2024, September 9). The Interaction Design Foundation. https://www.interaction-design.org/literature/topics/usability-testing#what_is_usability_testing?-0

[32] Aparna, K. S. (2024, October 24). Smarter, Faster, better: The rise of AI in usability testing and what it means for UI/UX designers. Aufait UX. https://www.aufaitux.com/blog/ai-powered-usability-testing-automation/

[33] Gupta, Y. (2012, December 23). Various Special Types of Tests Performed by Software Testing Engineers. Software Testing Genius. https://www.softwaretestinggenius.com/various-special-types-of-tests-performed-by-software-testing-engineers/

[34] TestBytes. (2019, June 27). Smoke Testing - Explanation With Example. Testbytes. https://www.testbytes.net/blog/smoke-testing-explanation-example/

[35] Jest Documentation, 2024 article

[36] Miller, A. (2021, February 23). Performance Testing, Load Testing & Stress Testing Explained. BMC Blogs. https://www.bmc.com/blogs/load-testing-performance-testing-and-stress-testing-explained/

[37] Pinto-Santos, J. A., Rodríguez-Borbón, M. I., & Rodríguez-Medina, M. A. (2021). An Introduction to the Accelerated Reliability Testing Method: A Literature Review. An Introduction to the Accelerated Reliability Testing Method: A Literature Review. https://www.researchgate.net/publication/355484479_An_Introduction_to_the_Accelerated_Reliability_Testing_Method_A_Literature_Review

[38] Elazar, E. (2018, April 23). What is User Acceptance Testing (UAT) - The Full Process Explained. Panaya. https://www.panaya.com/blog/testing/what-is-uat-testing/

[38] Savin, R. (2016, March 27). The Software Testing Life Cycle: Test Execution. QA training. How to Become a Software Tester (QA Engineer). https://www.qatutor.com/qa-course/lecture-5-the-software-testing-life-cycle/test-execution

[39] Andy Knight. (2020, July 7). Arrange-Act-Assert: A Pattern for Writing Good Tests. Automation Panda. https://automationpanda.com/2020/07/07/arrange-act-assert-a-pattern-for-writing-good-tests/

[40] Test lifecycle | Grafana k6 documentation. (2024). Grafana Labs. https://grafana.com/docs/k6/latest/using-k6/test-lifecycle/

[41] Talent. (2024, January 24). Best Practices for Effective Software Testing in Agile Development. DEV Community. https://dev.to/talenttinaapi/best-practices-for-effective-software-testing-in-agile-development-i7n

[42] Schwering, R. (2023). Defining test cases and priorities | Articles. Web.dev. https://web.dev/articles/ta-test-cases

[43] Bose, S., & Felice, S. (2024, October 25). Test Environment: A Beginner's Guide. BrowserStack. https://www.browserstack.com/guide/what-is-test-environment

[44] Schwartz, C. (2024, April 22). Automated Regression Testing: 2024 Guide. Automated Regression Testing: 2024 Guide. https://www.leapwork.com/blog/automated-regression-testing-guide

[45] Rahman, I., & Mabood, W. (2023, February 22). Maintaining Code Quality with Amazon CodeCatalyst Reports | Amazon Web Services. Amazon Web Services. https://aws.amazon.com/blogs/devops/maintaining-code-quality-with-amazon-codecatalyst-reports/

[46] Bhardwaj, D. (2024, February 5). What is Test Data in Software Testing. DEV Community. https://dev.to/devanshbhardwaj13/what-is-test-data-in-software-testing-4eam

[47] Test Data Management TDM | Test Data Generation. (2024). Genrocket.com. https://doi.org/10821579184/g5E7CNq-8I8DELDjkKgo

[48] Widder, D. G., & Goues, C. L. (2024). What Is a "Bug"?. Communications of the ACM. https://doi.org/10.1145/3662730

[49] Aedah Abd Rahman, & Nurdatillah Hasim. (2015). Defect Management Life Cycle Process for Software Quality Improvement. Zenodo (CERN European Organization for Nuclear Research). https://doi.org/10.1109/aims.2015.47

[50] Testim. (2022, March 25). Jest Testing Tutorial: 5 Easy Steps - Testim Blog. AI-Driven E2E Automation with Code-like Flexibility for Your Most Resilient Tests. https://www.testim.io/blog/jest-testing-a-helpful-introductory-tutorial/

[51] Hamilton, T. (2024, April 5). Severity & Priority in Testing: Differences & Example. Www.guru99.com. https://www.guru99.com/defect-severity-in-software-testing.html

[52] Nindel-Edwards, J., Steinke, G., & Steinke, J. (2006). A full life cycle defect process model that supports defect tracking, software product cycles, and test iterations. Communications of the IIMA, 6. https://scholarworks.lib.csusb.edu/cgi/viewcontent.cgi?article=1290&context=ciima

[53] Alshazly, A. A., Elfatatry, A. M., & Abougabal, M. S. (2014). Detecting defects in software requirements specification. Alexandria

Engineering Journal, 53(3), 513–527. https://doi.org/10.1016/j.aej.2014.06.001

[54] Wilkes, A. (2023, September 13). The Software Defect Life Cycle. Launchable. https://www.launchableinc.com/blog/the-software-defect-life-cycle/

[55] Cheatham, J. (2020, May 19). Snapshot Testing: Benefits and Drawbacks. SitePen. https://www.sitepen.com/blog/snapshot-testing-benefits-and-drawbacks

[56] Ahmad, N. (2024, February 20). How To Mark Bugs Using Real-Time Testing? | One-Click Bug Logging | Real-Time Testing | Part II. LambdaTest. https://www.lambdatest.com/blog/bug-severity-and-priority/

[57] Serban, C., & Vescan, A. (2019). Predicting reliability by severity and priority of defects. https://doi.org/10.1145/3340495.3342753

[58] Satyabrata, J. (2021, October 15). Defect Report in Software Engineering. GeeksforGeeks. https://www.geeksforgeeks.org/defect-report-in-software-engineering/

[59] Garousi, V., Ergezer, E. G., & Herkiloğlu, K. (2016). Usage, usefulness and quality of defect reports. https://doi.org/10.1145/2915970.2916009

[60] Pavlova, I. (2024). Best Practices for Effective Bug Reporting in Software Testing. TestDevLab Blog. https://www.testdevlab.com/blog/best-practices-for-effective-bug-reporting-in-software-testing/

[61] Surya. (2024). How to Write a Good Defect Report in Software Testing. F22 Labs. https://www.f22labs.com/blogs/how-to-write-a-good-defect-report-in-software-testing/

[62] Ganguly, S. (2024, July 26). How to Write a Good Defect Report? BrowserStack. https://www.browserstack.com/guide/how-to-write-a-good-defect-report/

[63] The CTO Club. (2024, October 24). 20 Best Defect Tracking Tools Reviewed for 2024. https://thectoclub.com/tools/defect-tracking-tools/

[64] BairesDev. (n.d.). Efficient Defect Reporting & Tracking Guide. https://www.bairesdev.com/blog/defect-reporting-and-tracking/

[65] Launchable Inc. (2023, January 17). Mastering Defect Tracking: Tools for Software Quality Assurance.

[66] Daily.dev. (2024, September 11). Defect Tracking Best Practices for Software QA.

[67] Colantonio. (2024, September 10). What is Automation Testing (The Ultimate Guide 2024). Test Guild. https://testguild.com/automation-testing/

[68] Martinez, D. (2024, March 27). 5 Benefits of Automated Testing and Why It Is Important. Telerik Blogs. https://www.telerik.com/blogs/5-benefits-automated-testing-why-important

[69] Kanade, V. (2024, May 14). Test Automation Approaches, Methodologies, Tools, and Benefits. Spiceworks Inc. https://www.spiceworks.com/tech/devops/articles/what-is-test-automation/

[70] Docheva, T. (2024, July 17). The Complete Guide to Test Automation: Best Strategies, Frameworks, and Tools. GetXray. https://www.getxray.app/blog/the-complete-guid

e-to-test-automation-best-strategies-frameworks-and-tools
[71] Adamik, J. (2024, November 13). Test Automation Strategy: Best Practices & Examples. Netguru. https://www.netguru.com/blog/test-automation-strategy-practices-and-examples
[72] Gadhiya, M. (2024, July 29). 17 Most Popular Unit Testing Frameworks in 2024. LambdaTest. https://www.lambdatest.com/blog/unit-testing-frameworks/
[73] What is the importance of communication in software testing? (2023, December 5). LinkedIn. https://www.linkedin.com/advice/3/what-importance-communication-software-testing-ypqxc
[74] Hilton, J. (n.d.). Build, Refactor, Repeat. https://jonhilton.net/refactoring/
[75] Vaidya, N. (2023, February 20). Jest Framework Tutorial: How to Use It? BrowserStack. https://www.browserstack.com/guide/jest-framework-tutorial
[76] Taleb, M. (2024, January 6). JavaScript Unit Testing Frameworks in 2024: A Comparison. Raygun Blog. https://raygun.com/blog/javascript-unit-testing-frameworks/
[77] Casariego. (2020, September 6). Regression Testing. Jorge Casariego. https://jorgecasariego.github.io/regression-testing/
[78] Kitakabee. (2023, February 10). Automated Regression Testing: A Detailed Guide. BrowserStack. https://www.browserstack.com/guide/automated-regression-testing
[79] Schwartz, C. (2024, April 22). Automated Regression Testing: 2024 Guide. Leapwork. https://www.leapwork.com/blog/automated-regression-testing-guide
[80] Convergence, I. (2024, July 4). 4 Ways to Minimize Test Automation Maintenance. IT Convergence. https://www.itconvergence.com/blog/4-ways-to-minimize-test-automation-maintenance/
[81] Parnel, T. (2024, November 20). How to Write Test Cases With Automation Tools. Qodo. https://www.qodo.ai/blog/how-to-write-test-cases-with-automation-tools/
[82] Ahamed, I. (2023, December 9). Automated Test Execution Tutorial: Comprehensive Guide With Best Practices. LambdaTest. https://www.lambdatest.com/learning-hub/automated-test-execution
[83] Veerappan, K. (2023, May 3). Measuring the Effectiveness of Test Automation. Zuci Systems. https://www.zucisystems.com/blog/measuring-the-effectiveness-of-test-automation/
[84] Ahamed, I. (2024, February 16). Use case testing tutorial: Comprehensive guide with best practices. https://www.lambdatest.com/learning-hub/use-case-testing

APPENDIX B: JEST INSTALLATION

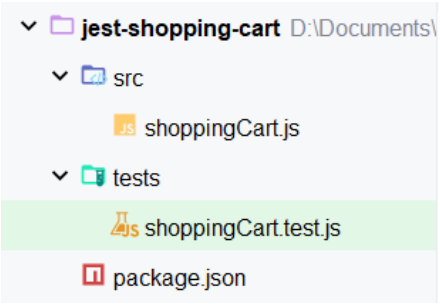
This section provides a step-by-step guide to configuring Jest for a JavaScript project. To illustrate its use, we will test a program designed to calculate the total price of items in a shopping cart, including applications of discounts.

STEP 1: Project Setup

Start by initializing a new package.json file using npm.

```
npm init -y
```

Create a basic structure for your project:



STEP 2: Install Jest

Install Jest using a package manager like npm:

```
Neil@LENOVO MINGW64 /d/Documents/CS-3-1/CS-312/jest-shopping-cart
$ npm install --save-dev jest
```

Add a test script to your package.json:

```
"scripts": {
 "test": "jest"
}
```

STEP 3: Write Your Program

For this setup, a JavaScript file names shoppingCart.js is placed inside the src/ folder:

```
function calculateTotal(cart, discount = 0) {
   if (!Array.isArray(cart)) {
       throw new Error("Cart must be an array.");
   }

   const total = cart.reduce((sum, item) => {
       if (!item.price || !item.quantity) {
           throw new Error("Each item must have a price
and quantity.");
       }
       return sum + item.price * item.quantity;
   }, 0);

   return total - (total * discount) / 100;
}

module.exports = calculateTotal;
```

This function calculates the total price of items in a cart, applies a discount when provided, and validates input data.

STEP 4: Write Test Cases

Create a test file inside the tests/ folder. In this case a JavaScript file names shoppingCart.test.js is placed inside the tests/ folder:

```
const calculateTotal = require('../src/shoppingCart');
```

```
describe("Shopping Cart Tests", () => {
   test("Calculates the total without discount", () => {
       const cart = [
           { price: 100, quantity: 2 },
           { price: 50, quantity: 1 },
       ];
       const total = calculateTotal(cart);
       expect(total).toBe(250); // 100 * 2 + 50
   });

   test("Calculates the total with a discount", () => {
       const cart = [
           { price: 200, quantity: 1 },
           { price: 150, quantity: 2 },
       ];
       const total = calculateTotal(cart, 10);
expect(total).toBe(450); // (200 + 300) - 10%
   });

   test("Throws an error for invalid cart input", () =>
{
       expect(() =>
calculateTotal("not-an-array")).toThrow(
           "Cart must be an array."
       );
   });

   test("Throws an error for invalid item format", () =>
{
       const cart = [{ price: 100 }];
       expect(() => calculateTotal(cart)).toThrow(
           "Each item must have a price and quantity."
       );
   });
});
```

STEP 5: Run Tests

Use the npm test command to run the tests:

```
Neil@LENOVO MINGW64
$ npm test

> jest-shopping-cart@1.0.0 test
> jest

 PASS  tests/shoppingCart.test.js
  Shopping Cart Tests
    √ Calculates the total without discount (6 ms)
    √ Calculates the total with a discount (1 ms)
    √ Throws an error for invalid cart input (15 ms)
    √ Throws an error for invalid item format (2 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        1.298 s

Ran all test suites.
```

APPENDIX C: API TESTING PROGRAM

The open-source API used for the API testing used in this project is as follows:

Site: https://www.coinlore.com/
API URL: https://www.coinlore.com/

This API serves as the API for demonstrating Jest's testing features, such as Snapshot Testing.

JavaScript file composed mainly of two async functions for simulating API calls.

```
const axios = require('axios');

const BASE_URL = 'https://api.coinlore.net/api/';

const getCryptos = async () => {
   try {
       const response = await
axios.get(`${BASE_URL}tickers/`);
       return response.data;
   } catch (error) {
           throw new Error('Error fetching data from
Coinlore');
   }
};

const getCryptoById = async (id) => {
   try {
        const response = await
axios.get(`${BASE_URL}coin/markets/?id=${id}`);
       return response.data;
   } catch (error) {
           throw new Error('Error fetching data from
Coinlore');
   }
};

module.exports = { getCryptos, getCryptoById };
```

This file contains unit tests for functions interacting with the Coinlore API, using Jest to ensure proper functionality and error handling. It tests both the fetching of cryptocurrency data and specific coin details while employing mock implementations of axios to simulate API responses.

```
const    {    getCryptos,    getCryptoById    }    =
require('./coinlore');
const axios = require('axios');

jest.mock('axios');

describe('Coinlore API', () => {
   afterEach(() => {
       jest.clearAllMocks();
});

    test('should fetch the list of cryptocurrencies',
async () => {
       const mockData = {
           data: {
               coins: [
                   { id: 90, name: 'Bitcoin', symbol:
'BTC', price_usd: 45000 },
                   { id: 2710, name: 'Ethereum', symbol:
'ETH', price_usd: 3000 }
               ]
           }
       };

       axios.get.mockResolvedValue(mockData);

       const result = await getCryptos();
```

```
        expect(result).toEqual(mockData.data);

expect(axios.get).toHaveBeenCalledWith('https://api.coin
lore.net/api/tickers/');

        expect(result).toMatchSnapshot();
    });

     test('should fetch a specific cryptocurrency by ID',
async () => {
        const mockData = [
               { id: 90, name: 'Bitcoin', symbol: 'BTC',
price_usd: 45000, market_cap_usd: 850000000000 }
        ];

        axios.get.mockResolvedValue({ data: mockData });

        const result = await getCryptoById(90);
        expect(result).toEqual(mockData);

expect(axios.get).toHaveBeenCalledWith('https://api.coin
lore.net/api/coin/markets/?id=90');

        expect(result).toMatchSnapshot();
    });

    test('should throw an error when Coinlore API fails',
async () => {
            axios.get.mockRejectedValue(new  Error('Error
fetching data from Coinlore'));

        try {
            await getCryptos();
        } catch (e) {
              expect(e).toEqual(new Error('Error fetching
data from Coinlore'));
        }
    });
});
```

Using npm test or jest to run the test. This will be the expected output.

```
rouxvick@rouxvick-Aspire-A315-24P:~/WebstormProjects/jestTesting$ jest
 PASS  ./coinlore.test.js
  Coinlore API
    ✓ should fetch the list of cryptocurrencies (4 ms)
    ✓ should fetch a specific cryptocurrency by ID (2 ms)
    ✓ should throw an error when Coinlore API fails (1 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   2 passed, 2 total
Time:        0.268 s, estimated 1 s
Ran all test suites.
```

APPENDIX D: SYSTEM AND USER ACCEPTANCE TESTING PROGRAM

The e-commerce product store application used for system and user acceptance testing in this project is as follows:

- **Repository Name:** mern-crash-course
- **Repository URL:** https://github.com/burakorkmez/mern-crash-course
- **Creator/Owner:** burakorkmez

This repository serves as the foundational codebase for demonstrating various testing strategies, including system testing and user acceptance testing.