Journal of Information Systems Engineering and Management
2024, 9(2)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Test-Driven UI Development with Jasmine, Karma, and Protractor

Manasa Talluri
Independent Researcher, USA

ARTICLE INFO

Received: 02 Apr 2024

Accepted: 28 Jun 2024

ABSTRACT

This study attempts to expose and compile testing-related data from Stack Overflow, emphasising it as a useful tool for professionals looking for solutions and direction. This gives usability tests a chance to become better. The current study addresses usability issues, with a focus on the usability of individual user interface (UI) components designed for online interfaces. A multitude of technologies make up today's continuous delivery (CD) pipeline, which guarantees that the best possible product is produced. A new feature undergoes many testing stages as it progresses from conception to production. Nevertheless, the majority of this testing has to be automated in order for the CD pipeline to continue operating efficiently. Automation infrastructure is necessary for user interface, integration, load, and unit testing. We utilise a variety of supporting technologies, including Gerrit, Jenkins, Gulp, Protractor, Jasmine, Docker, and Kubernetes, to automate testing for each release, run lengthy soak tests, and facilitate quick iterations on deep performance tuning tasks involving intricate cluster configurations. To ensure that every release maintains high quality while consistently providing value to our clients, this paper demonstrates how to incorporate various testing methods into the CD process.

Keyword: Stack Overflow, UI Components, Usability Tests, Jasmine, Docker, And Kubernetes, Continuous Delivery (CD), Customers.

INTRODUCTION

Nowadays, almost all services and apps on all platforms are either looking for methods to reduce update cycle time or are already regularly providing customers with updates. A widespread absence of high-fidelity automated testing and the separation of development and testing resources hampered many products in this endeavour until recently [1, 2]. Because of these two issues, total test cycles were often long and often lacking. Including testing resources in the development team and combining testing and development into a single continuous delivery flow is a preferable approach.

A crucial step in the software development process is software testing. Early defect and bug detection in software development boosts customer happiness and confidence, reduces errors and failure, identifies security vulnerabilities, improves product quality, aids in scalability, and saves money in a variety of scenarios [2, 3]. According to studies, practitioners have often received insufficient training in software testing, despite the significance of software testing. When practitioners have concerns about testing, they search the internet for information. In the field of computer science, Stack Overflow (SO1) is a well-known forum. SO is a practitioner question-answer forum that was established in 2008 with the goal of fostering peer conversations and information exchange [3].

In order to keep consistent, adhere to the DRY (don't repeat yourself) paradigm, and save production costs, it is generally accepted practice to break up user interfaces into modular components that may be reconfigured to generate various design patterns. Because it takes a lot of work to learn and adjust to a new software tool stack (e.g. workflow and build procedures, syntax, patterns, methodology, development environment, etc.), some programmers still do not follow this tried-and-true method when making small websites. However, same ideas are used in sophisticated online applications [2, 4], otherwise they would be difficult to use and maintain. Additionally, in order to remain relevant in a fiercely competitive global market, it is crucial to be able to quickly adapt new or current information systems to make them more effective, since business demands are always changing [3, 4]. Instead, then completely redesigning the system every few years, segmentation seems to be the answer to addressing this issue in a more methodical and practical manner.

Application testing is an essential component of development throughout all stages, regardless of how the business objectives are serviced. The final output depends on ensuring that all of the needs of the earlier processes are correctly carried out, whether this is done during the design, programming, conceptual development, and specification clarification phases. As a result, testing is not only one phase in the development cycle [3, 5], but it may and should be carried out by all participants to ensure high-quality products and prevent misunderstandings and poor communication between members.

With the rising popularity of remote project cooperation, this is becoming more and more crucial.Additionally, the absence of different testing methods often leads to greater development costs since several processes must be completed in order to comply with the final testing assessment. For instance, the quality assurance engineer will alter (a) the element's requirements, (b) the design, and (c) the source code if, during functional testing, [2, 3], they discover an issue brought on by an inefficient text size. This leads to the obvious conclusion that software quality will increase with the number of tests conducted.

Usability testing, better known as a subfield of the user experience (UX) area of expertise, continues to be one of the most important topics in the world of software testing. The usability of online apps is a crucial component. It should go without saying that a software product should be tested often with prospective users, but this is not a usual practice and typically calls for skilled UX engineers and testers, whether at the idea or realisation stages. However, due to the large body of knowledge that provides standards, conventions, and recommendations, usability factors—like accessibility, for instance—can be easily automated and included into the development process [3, 4].

The bottom-up strategy of accessibility-driven development, for instance, requires that the product be beneficial to individuals with impairments, novice users, people from diverse backgrounds and languages, and many government organisations strive to follow this philosophy. Given the high degree of human-software compatibility offered, it stands to reason that the typical user would only gain from the comprehensive and thoughtful user experience [3, 4]. Similar to this, customisation may increase user happiness and usability by offering distinct user groups unique designs and experiences. To allow for a wider range of public customisation, several interfaces now provide other themes, such as dark, bright, or high contrast.

Between 2009 and 2014, Kochhar conducted research on the subject of extracting testing knowledge from Stack Overflow. The study produced information on significant debate points, changes throughout time, and technological difficulties developers confront. Testing frameworks, databases, threads, forms, builds, etc. are among the topics that are often discussed on SO, according to the research's findings. Databases, client servers, and testing frameworks were the "hot topics" within these categories throughout the study period [3, 5]. Additionally, there has been a significant increase in queries related to mobile development.

The SO subtopic of software engineering is called release engineering (re-leng). Their study focusses on the kinds, prevalence, and answerability of release engineering-related problems. The primary conclusions indicate that continuous integration/continuous deployment (CI/CD) is the subject of the majority of enquiries. Additionally, they uncover a negative relationship between topic popularity and difficulty, indicating that challenging subjects are less likely to be well-liked [3–6].

investigated the methodological specifications that determine "best practices." Their study examines several statistical techniques that empirically identify "best practices" and offers a framework for doing so [3, 7].

TESTING TOOLS & FRAMEWORKS

For Angular testing, we provide a variety of frameworks and testing tools. Here are a few examples of popular tools and frameworks:

1.1 Karma

Karma is a test runner that was created by Vojta Jina. This project's original name was Test-acular. It allows us to quickly and automatically run JavaScript unit tests on actual browsers since it is not a test framework. Karma was created for AngularJS because, prior to Karma, web-based JavaScript developers lacked an automated testing tool [5, 8]. However, developers may use Karma's automation to execute a single command and see whether a whole test suite succeeded or failed.

1.2 Jasmine

Pivotal Labs creates the behavior-driven development framework Jasmine, which is used to test JavaScript code. Prior to the Jasmine framework's active development, Pivotal Labs created JsUnit, a comparable unit testing framework with an integrated test runner [3, 6]. By incorporating the SpecRunner.html file in the Jasmine tests, or by using it as a command-line test runner, the browser tests may be executed. Additionally, it may be used with or without Karma.

We begin this article by outlining our approach to product development and the importance of automated testing to our continuous delivery (CD) pipeline. In-depth explanations of the tools and methods we use to automatically test every build—even throughout the code review process—follow [3, 7]. Lastly, we'll outline our strategies for enhancing our test coverage and infrastructure going forward as our product becomes more well-known, complex, and essential to our operations.

1.3 Lean Continuous Delivery

It's crucial to comprehend how our team creates items before discussing how we test them, since this clarifies the kind of result we want to attain and the rationale for our testing methods. For two primary reasons, our approach to product creation incorporates Design Thinking, Lean and Agile creation, and other best practices.

First of all, we are a business that needs to maximise profits. Profit is influenced by a lot of indirect factors, but we may directly affect it by eliminating needless work (e.g., avoid constructing stuff people won't need and avoid delivering broken code). Our second goal is to create features that users would love [3, 6]. Revenue is increased by delightful items, and our total profit will increase the sooner we reach this degree of perfection. When creating goods, we combine several facets of our concept (Figure 1). We divide them into aspects, skill sets, and principles [5, 6]. The guiding principles are:

Sharing skills: We think that sharing skills helps us develop as a team [5, 6]. Our team is poor if each member is the sole one with knowledge on a particular topic. In addition to having a lower overall pace, weak teams are less able to intuitively comprehend the appropriate trade-offs while implementing new features. Teaching a skill also improves one's own proficiency.

- **Lean:** We think that persistently removing waste is the greatest method to quickly provide value to our market. We can increase our velocity, market accuracy (i.e., [7, 8], providing the correct item at the right time), and customer happiness by getting rid of waste.
- **Design thinking:** Whether they are users, requestors, benefactors, or everyone else our product effects, we think that the greatest solutions arise from understanding them, and that design thinking is the best way to do this. Our internal value flow is strongly impacted by design thinking (Figure 2) [5, 6]. A higher-quality product that people love using is directly the result of the customer viewpoint that design thinking gives us [3].
- **"Us":** All of the universe is referred to as "Us"; everything affects our team, product, market, etc. Everything encompasses both known and unknown information. Everything is essential. But our team is most affected by four factors in particular: our market, our corporate culture, the history of Agile development, and our people and their experiences [6–7].

Fig. 1 Some of our philosophical stances. [25]

Our team's diverse functional specialisations are known as skillsets:

- **Design:** This characterises our product's form and functionality. This skill set takes into account the user experience, how those features work (i.e., the features of a feature), which features to install, and which to forego. The whole experience that a product offers is its design, while its user interface is its exterior [7, 8]. In other words, an unattractive design cannot be fixed by even the most beautiful user interface.
- **Architecture:** What design is to our product's outside; architecture is to its inside. It is the product's internal organisation [8, 9]. Choosing which technologies and approaches (like reactive) to use and how to combine them to provide the foundation for implementing new functionality are examples of architectural decisions.
- **Implementation:** This is code writing itself, regardless of other abilities. That is to say, it is the how of creating a new feature rather than the why or what, and it involves producing code that follows best practices.
- **UI/UX**: Our direct human connection is described here. Once again, not every person who uses and is impacted by our product is doing so via the online user interface.mSimilar to design, user interface and experience (UX) take into account all user concerns as well as the exterior repercussions of our product [23, 24]. Because badly organised features result in inferior experiences, UI/UX is intimately related to design and architecture. UI/UX encompasses more than simply GUI functionality; it also covers file formats, log messages, and APIs [8].
- **Performance:** We consider our product's computational efficiency to be its performance. Raw numbers are the main focus of performance: how many, how little, and how quickly. When it comes to computers, people demand exceptional performance, therefore we strive to create products that are fun to use.
- **Verification:** Verification guarantees that we create what we set out to create. "Did we build it correctly?" is the question that verification asks. (As opposed to validation, which responds to the question, "Are we building the right thing?") [9, 10]. This not only indicates that we create goods that are sufficiently free of bugs but also that they perform as promised—neither more nor less.

TESTING TECHNOLOGIES

Our online user interface is based on Bootstrap and AngularJS. Our servers are accessed by the user interface (UI) using REST APIs, which return data in JSON format. Although this is a fairly standard configuration for web applications nowadays, it directs technological choices due to their particular emphasis on certain features of our product [23]. Other technologies that do the same task may be chosen for non-AngularJS/Bootstrap UI stacks [11] (for instance, Selenium itself is a fairly full-featured automated testing tool that works with a wide range of data and UI technologies).

- **Code Review:** Jenkins builds the product once a developer submits code for review to make sure the code doesn't interfere with the build. This build also guarantees that all unit tests pass since we utilise them throughout our product [11, 12].
- **Post-merge:** Jenkins produces the product for deployment when the code review is accepted and submitted via Gerrit. The build fails and the artefacts are not submitted to the Docker Registry if any of the tests are

unsuccessful. These tests are limited to UI tests utilising a simulated environment in order to save build time [13].

- **Pre-deployment:** A comprehensive suite of automated tests makes sure that nothing has gone wrong with the product before containers are sent to Docker Registry. Using Gulp and Protractor, these pre-deployment tests fully test the deployment of all containers, including the data simulator and UI testing components [13, 14].

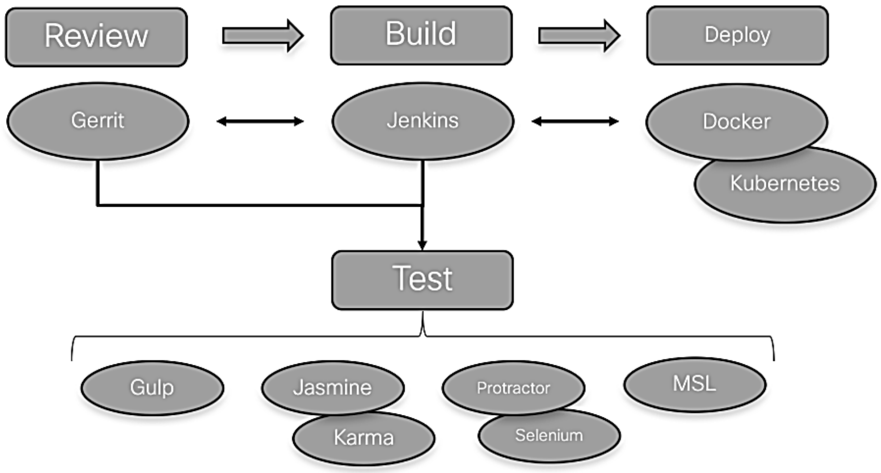

Fig. 2 Create and Test a Pipeline (chosen technologies). [15]

We utilise Gerrit to handle our code reviews and Jenkins as our build system orchestrator. Jenkins is triggered by Gerrit when a developer wishes to submit a patch for review in order to make sure the change doesn't damage the build [22]. In addition to unit tests, the build process incorporates a few short-lived automated tests to increase coverage before a code review request is made.

Docker containers are used by all of our components. These consist of frontend web servers, database servers, and proprietary product components. Containers from the build system are distributed to internal and external deployments using Docker Registry [13]. We utilise Kubernetes pods and services to manage our component containers in order to regulate our clustered deployments. Since Kubernetes offers automated cluster management, production-grade automated E2E testing may take place as part of the build process, even if it isn't exactly a testing solution.

In order to produce large lead levels for stress testing installations, we developed a highly scalable traffic simulator. Verification tests are also carried out using it in setups with lesser volumes. In order to mimic vast volumes of historical data, the traffic simulator creates the proprietary data we get via our southbound interface (the data collection portion of our system), enters the data straight into our database, and uses the northbound (web) interface to execute queries and user interface calls [14]. In order to enable automatic end-to-end testing and confirm that certain data passes through the system as intended, we will also eventually integrate a web interface to this simulator [20, 21].

Our automated tests are run as part of our JavaScript build process, which is orchestrated by Gulp. Gulp works similarly to Make, Maven, or SBT for us. But it does more than simply package our JavaScript files (cleanup, debug code removal, minification, obfuscation, [14,16], etc.); it also runs our automated tests by launching the test tools first, followed by the tests.

A framework for creating and executing test-driven development (TDD) and behavior-driven development (BDD) tests for web user interfaces is offered by Jasmine. It helps our commitment to BDD/TDD by enabling parallel execution and performing certain subsets of tests [16]. Additionally, we found it to be quite strong but user-friendly.

Karma is a test runner for Jasmine. Karma performs each Jasmine test and delivers the results once developers choose a JavaScript engine to test against (such as PhantomJS, Chrome, or Firefox). Karma is the program that actually executes the BDD/TDD tests, whereas Jasmine is the BDD/TDD test environment.

Typically, unit testing simply evaluates one piece of functionality in one manner. To thoroughly verify a single piece of functionality, many unit tests are combined. This is the extent of what we believe to be unit testing; it is more likely to be some kind of integration testing. For our UI unit tests, we utilise Gulp, [17], Karma, and Jasmine, and for our server code, we use Scala Test [20, 21].

The best method for ensuring that the whole system will function when it is deployed is end-to-end testing, or E2E. E2E is the method by which we verify functionality from data intake to presentation, while unit testing and mocked-data testing guarantee that parts of the product operate independently. In order to put a deployment under heavy pressure, our team has created a scalable data simulator that can provide data to our ingest endpoints and strike our REST APIs. Real data sources are not used in automated testing because they are too sluggish and difficult to manage [11, 16]. E2E testing' main benefit is that they really put the complete product to the test. E2E tests do, however, have a number of drawbacks:

- Because test data and application flow are extensive and intricate, tests are delicate and difficult to manage.
- Because the whole system has to be operational, tests take longer to complete. Because data must be processed by actual solution components, there is also an end-to-end lag time to get it through the system [16].
- All of the E2E build, deploy, and test time would have been saved if a component test (non-E2E) had discovered the flaw in a single component that would have eventually caused an E2E test failure.
- Needs more research since defects discovered during the test might originate from anywhere in the system. Since test failures may not be caused by the feature being tested, developers must look into the underlying cause.

We understood early on that in order to integrate the UI unit tests into our build process without having to launch ephemeral virtual machines (VMs) with actual (virtual) screens, we required a headless browser. Nevertheless, we then discovered that part of our UI JavaScript (JS) was improperly executed by the technology we had selected (PhantomJS) [16, 17]. This indicated that the tests were unsuccessful and would never be successful. Therefore, in order for Selenium to run a reliable JS engine, we switched to utilise a genuine Chrome browser that was running with xvfb.

Remember that AngularJS is the foundation of several of our solutions, including Protector [19, 20]. Even if the new component otherwise results in no philosophical modifications to our construction (i.e., it behaves the same way, [17], but is implemented slightly differently), this means that any decision to move away from AngularJS also entails replacing that portion of our test infrastructure and rewriting all the tests that assume AngularJS is part of our architecture.

1.4 Example Test Script

This test script demonstrates how Protractor, MSL, and Jasmine collaborate to test the functioning of the product. This test involves logging in and verifying that the page that loads right away has some data [18, 19]. The describe block's comments provide the test case steps.

```
// Requires the MSL client API
var msl = require('msl-client');

console.log("Executing Mock page....");
describe("Mock Signin", function () {
    "use strict";
    var subscriberCount = element.all(
        by.css('div[class*="table-responsive"] tr[pagination-id="search-results"]'));

    describe("Mocking UI Test", function () {
        /* Step-1 Go to Signin page
         * Step-2 Enter user name
         * Step-3 Enter password
         * Step-4 Click submit button
         * Verify 10 subscribers exist in first page */

        var mockResponseObj = {};
        mockResponseObj.requestPath = "/api/authenticate";
        mockResponseObj.responseText = [{"username":"tester","password":"testme"}];
        mockResponseObj.contentType = "application/json";
        mockResponseObj.statusCode = 200;
        mockResponseObj.delayTime = 0;
        mockResponseObj.header = {"X-App-Token" : "r4nd0mT3x7"};

        it("should sign in", function () {
            // signin mock response
            msl.setMockRespond("localhost", 8000, mockResponseObj);

            // open a browser and go to sign-in page
            browser.get('http://localhost:8000');

            // sign in
            element(by.id('username')).sendKeys('user');
            element(by.id('passphrase')).sendKeys(‘password’);
            element(by.css("form[ng-submit='signInCtrl.submit()'] button")).click();
        });

        // make sure it has what it should have immediately after signing in
        it ("should have 10 subscribers in first page", function () {
            expect(subscriberCount.count()).toBe(10);
        });
    });
});
```

CONCLUSION

We can only provide high-quality updates by integrating automated testing into our pipeline, yet our clients want the time-to-fix immediacy that CI/CD can offer. Modern online applications that are high-quality, high-value, and reliable are difficult to build; automated testing framework and individual tests alone are challenging.

Nevertheless, it's an essential step in providing our consumers with ongoing product upgrades, and we couldn't continue to provide the responsiveness they want without it. We're expanding our automated QA system in two ways now that the necessary components are in place. First, more tests are constantly needed to enhance code coverage of current features and introduce new ones.

We don't just mean more tests when we say "more tests." A crucial component of our pipeline is a high-performance simulator that can only produce random data. Because of this, E2E testing that assesses the precision of the data shown in the user interface is not feasible. Although it may not be the correct data, we are aware that we have some. Additionally, our simulator can put a deployment under high demand, but we lack coordinated UI or E2E tests to verify that the system reacts as anticipated under that stress.

Second, integration tests and soak tests are additional testing methods that we may use in our current workflow. We could, for instance, test the database-to-frontend interface separately for integration testing. Currently, only E2E testing exercises this interface, and a failure will only be detected if it results in a UI-layer failure. Additionally, any fuzzing or other heavy interface-targeted testing is prohibited while relying on E2E testing.

We might study the system over a period of hours, days, or weeks by subjecting it to significant loads using soak tests. It still uses automated testing to ensure that the data feeds and frontend REST requests are realistic, even though this is clearly not a part of the CI/CD testing. We're always looking for ways to enhance our development process so that we can provide our clients with the best possible quality and value. Given the flexibility and great capability of the framework, applying these lessons to our pipeline should be quite efficient.

REFERENCES

[1] Moses Openja, Bram Adams, and Foutse Khomh. “Analysis of Modern Release Engineering Topics : – A Large-Scale Study using StackOverflow –”. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). 2020, pp. 104–114.

[2] Stuart Bretschneider, Frederick J. Marc-Aurele Jr., and Jiannan Wu. ““Best Practices” Research: A Methodological Guide for the Perplexed”. In: Journal of Public Administration Research and Theory 15.2 (Dec. 2004), pp. 307–323. ISSN: 1053-1858.
[3] Miltiadis Allamanis and Charles Sutton. “Why, when, and what: Analyzing Stack Overflow questions by topic, type, and code”. In: 2013 10th Working Conference on Mining Software Repositories (MSR). 2013, pp. 53–56.
[4] Ardic Baris and Zaidman Andy. “Hey Teachers, Teach Those Kids Some Software Testing”. In: (2023). To appear in: IEEE/ACM 5th International Workshop on Software Engineering Education for the Next Generation (SEENG).
[5] Stack Overflow Traffic. Accessed: 17-06-2023.
[6] Sebastian Baltes et al. “SOTorrent: reconstructing and analyzing the evolution of stack overflow posts”. In: Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018.
[7] Ed. by Andy Zaidman, Yasutaka Kamei, and Emily Hill. ACM, 2018, pp. 319–330.
[8] Sita Shreeraman. ISTQB – Different Test Types. Accessed: 20-06-2023.
[9] Roshali Silva et al. “Effective use of test types for software development”. In: 17th International Conference on Advances in ICT for Emerging Regions, ICTer 2017 - Proceedings. Vol. 2018. 2017, pp. 7–12.
[10] Bootstrap. “Bootstrap - The world's most popular mobile-first and responsive front-end framework.” http://getbootstrap.com/ (accessed July 5, 2016).
[11] Cross, Nigel. 1982. “Designerly ways of knowing.” Design Studies 3.4: 221-27.
[12] Docker, Inc. “Docker - Build, Ship, and Run Any App.
[13] Gerrit. “Gerrit code review.” https://www.gerritcodereview.com/ (accessed July 5, 2016).
[14] Google, Inc. “Chrome.” https://www.google.com/chrome/ (accessed July 5, 2016).
[15] Gulp. “Gulp” https://www.npmjs.com/package/gulp (accessed July 5, 2016).
[16] Ho, Toby. January 9, 2015. “Headless browser testing with xvfb.” http://tobyho.com/2015/01/09/headlessbrowser-testing-xvfb/ (accessed July 5, 2016).
[17] Jasmine. “Jasmine: Behavior driven JavaScript.” http://jasmine.github.io/ (accessed July 5, 2016).
[18] Karma. “Karma - Spectacular test runner for JavaScript.” https://karma-runner.github.io/1.0/index.html (accessed July 5, 2016).
[19] Kubernetes. “Kubernetes - Production-grade container orchestration.” http://kubernetes.io/ (accessed July 5, 2016).
[20] Lightbend, Inc. “sbt - The interactive build tool.” http://www.scala-sbt.org/ (accessed July 5, 2016). MSL. “Mock service layer.” http://finraos.github.io/MSL/ (accessed July 5, 2016).
[21] Gediga, G., & Hamborg, K.-C. (1999). IsoMetrics: Ein Verfahren zur Evaluation von Software nach ISO 9241/10. In
[22] H. Holling & G. Gediga (Ed.), Evaluation stations for schung. Göttingen: Hogrefe.
[23] Gediga, G., Hamborg, K.-C., & Düntsch, I. (1999). The IsoMetrics usability inventory: An operationalization of ISO 9241-10 supporting summative and formative evaluation of software systems. Behaviour & Information Technology, 18(3), 151–164.
[24] Google. (2019). Material Design. Retrieved March 18, 2019.
[25] Green, T. R. G. (1990). Limited theories as a framework for human-computer interaction. In D. A. & M. J. Tauber (Ed.), Mental models human computer interaction.

Journal of Information Systems Engineering and Management
2024, 9(2)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article