

Spacetime.how Application Testing Project

Nick Forleo, Manish Shrestha, & Brian Stahl

SWENG 581: Software Testing

Group 9

#### **Section 1: Introduction**

**1.1 Test Project Name:**

"Spacetime.how Application Testing Project"

**1.2. Summary of the Rest of the Test Plan:**

This comprehensive test plan delineates the testing strategies and procedures to ensure the robust functionality of these vital features within the application. This test plan outlines the testing efforts for the spacetime.how application, focusing on its essential features related to time manipulation. The plan covers comprehensive range of elements, including feature descriptions, underlying assumptions, the chosen test approach, intricately designed test cases, specifics of the test environment, detailed insights derived from testing results, and thoughtful recommendations aimed at elevating the overall software quality of the spacetime.how application.

#### **Section 2: Feature Description**

The spacetime.how application encompasses a rich array of time-related functionalities designed to enhance user experience and facilitate efficient date and time manipulation. Key features include:

* Timezone Information Retrieval: The application allows users to access detailed information about different timezones, providing valuable insights into variations across geographic regions.
* Movement to Different Timezones: Users have the ability to seamlessly transition between various timezones, offering flexibility and adaptability to diverse global contexts.
* Positioning at the Start of Specific Time Units: The application facilitates precise positioning at the commencement of specific time units, enabling users to navigate time with accuracy.
* Date/Time Equality Check: Users can easily verify if two date/times are identical, ensuring precision in comparisons and operations related to date and time.
* 12-Hour Time Format Management: The application provides the functionality to set or retrieve time in the 12-hour format, catering to different preferences and conventions.
* Week Numbers, Fiscal Quarters, and Seasons Management: Users can efficiently manage and retrieve information related to week numbers, fiscal quarters, and seasons, facilitating organized time tracking and planning.
* Month Names and Date Object Manipulation: The application supports the manipulation of month names and other date objects, offering a versatile toolkit for users to tailor date representations according to their needs.

#### **Section 3: Assumptions**

**3.1 Test Case Exclusions:**

This subsection communicates that there were no deliberate exclusions in test case selection, and the testing team aimed to cover as much of the application as possible. The objective was not merely to fulfill a checklist of scenarios but to provide a robust and comprehensive examination of the application's functionalities. By eschewing deliberate exclusions, the team aimed to uncover potential vulnerabilities and intricacies that might be overlooked in a narrower testing scope.

**3.2 Test Tools, Formats, and Organizational Schemes:**

In this section, we delve into the diverse testing techniques employed to ensure the effectiveness and reliability of our application:

* Automated Test:
  + Strategically integrating automated tests was a cornerstone of our testing approach, emphasizing efficiency and repeatability. This strategic implementation facilitates the swift identification of issues, ensuring a consistently reliable testing outcome. Automated testing serves as a key catalyst in expediting the testing process, providing a robust foundation for identifying and addressing potential challenges throughout the development lifecycle.
* Input Space Partitioning:
  + Our team adopted a systematic approach by modeling input domains and partitioning function characteristics. This technique enhances the granularity of testing, allowing us to address various facets of the application's functionality systematically. By categorizing inputs into distinct partitions, we aim to ensure comprehensive testing coverage, identifying potential vulnerabilities and enhancing the overall resilience of the application.
* Graph-Based Testing:
  + The deliberate selection of a function with a complex decision tree underscores our strategic choice to achieve high code coverage. This method ensures a meticulous examination of intricate functionalities, contributing significantly to the overall robustness of the application. By navigating through the decision tree structure, we aim to uncover potential complexities and intricacies within the application's logic, ultimately enhancing its reliability.
* Exploratory Testing:
  + For a meaningful and user-centric evaluation, we identified the Guidebook tool as a valuable resource. The documentation's clarity in describing the purpose of functions and return types played a pivotal role in our testing approach. Given that users often rely on documentation to determine the most suitable function for a given problem, this tool was instrumental in verifying the accuracy and effectiveness of our documentation. This approach is particularly vital in user-centric applications, where clear and accurate documentation is paramount to guiding users effectively through the software's functionality.

#### **Section 4: Test Approach**

**4.1 Addressing Past Issues:**

When adding centuries to dates using add() method, the number of years added was not correct. For example, adding 1 century to the date Nov 30, 2000 resulted in Nov 30, 2100. However, if you wanted to add 2 centuries to Nov 30, 2000, it would still result in Nov 30, 2100. We acknowledged that this test would fail when writing our test cases so that it did not interrupt the rest of the test cases.

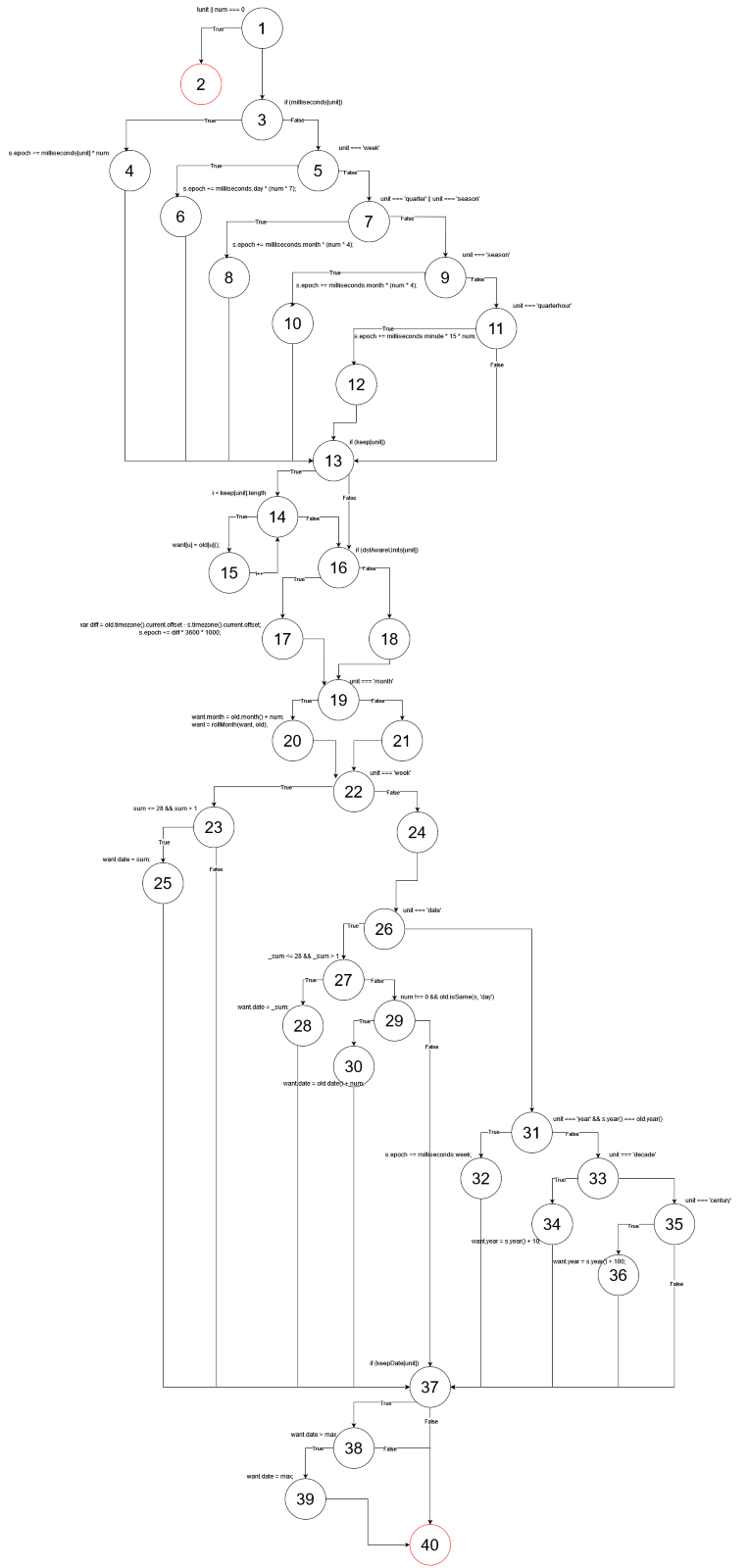
**4.2 Special Testing Considerations:**

The application has critical functionality around date manipulation and comparisons. As such, combinatorial test techniques were employed to cover permutations,

* Base Choice Coverage (BCC): Applied to most functions like add(), format() and, etc. BCC selected test cases to cover every base choice option within parameters at least once.
* All Combinations of Choices (ACoC): Used specifically for isSame() with its high complexity. ACoC combined all valid combinations of the date1/date2 inputs passed. This approach ensured a comprehensive assessment of the functionality, considering the diverse interactions and scenarios that could occur in practical use

**4.3 Test Strategy:**

* Automated Test:
  + In our testing approach, we prioritized the creation of automated tests that could be seamlessly executed. Automation offers several advantages, including efficiency, repeatability, and the ability to detect issues promptly. By writing automated tests we aimed to streamline the testing process and ensure a swift and consistent evaluation of the software's functionality. Automated testing allowed us to runour tests quickly, helping identify potential bugs, regressions, or performance issues in a timely manner. This not only accelerates the testing phase but also facilitates the integration of testing into continuous integration /continuous deployment, promoting a more agile and iterative development process.
* Input Space Partitioning
  + Input Space Partitioning is a systematic testing technique where the input domain of a system is divided into partitions or subsets, and test cases are derived from each partition. This approach helps ensure comprehensive coverage while minimizing redundancy in test scenarios. In our testing, we employed Input Space Partitioning by modeling the input domain and strategically partitioning function characteristics into distinct blocks. By doing so, we aimed to address various potential scenarios and ensure that our tests effectively covered different aspects of the input space. This method enhances the efficiency of our testing strategy, allowing us to focus on key areas and identify potential issues associated with different input conditions.
* Graph Based Testing
  + We selected the add() function from SpaceTime for graph-based testing. This function includes operations like normalizing the input time unit, estimating milliseconds to increment the epoch, adjusting components like month and date, handling things like daylight saving and leap years, and post-processing results. The complex control flow through these operations with branches and loops based on conditions could be modeled as a graph. Nodes would represent operations like normalizing units or incrementing the epoch. Edges would show transitions between nodes under different conditions like input values and date component values. By modeling it as a graph and applying coverage criteria like node coverage, we could systematically generate test cases to thoroughly cover different paths through this complex date manipulation logic. Graph-based testing would allow us to ensure full coverage of this tricky control flow. We selected node coverage as our testing criteria for the SpaceTime add() function because it requires executing every node in the control flow graph, ensuring systematic testing of all operations and decisions points represented as nodes - this exhaustive approach provides very thorough testing for the intricate date manipulation logic found in add(), as node coverage guarantees we fully cover every branch and statement by forcing execution of all possible paths through the graph model, delivering pervasive test coverage across the complex control flow and confidently validating the robustness of the tricky date math implemented in the SpaceTime add() function. This selection is rooted in several key factors and a thorough rationale that underpins our decision.
    - Comprehensive Code Inspection: This criterion ensures that every line of code is executed at least once during testing. This means that our testing efforts will meticulously scrutinize each statement within our application, leaving minimal room for undiscovered code paths or potential bugs. By achieving comprehensive code inspection, we can be confident that our application has been thoroughly assessed for defects.
    - Bug Detection: The goal is to uncover and eliminate software defects. Node coverage excels in this regard. By executing every statement in our codebase, we increase the likelihood of identifying latent bugs and/or unforeseen issues. Early bug detection reduces the chances of these issues surfacing in a production environment where they could lead to customer dissatisfaction and costly maintenance.
    - Risk Mitigation: Node coverage helps manage risks effectively. It ensures that all the core functionality and critical components of our application are rigorously tested. This way, we address potential vulnerabilities or weak points in our code, which is vital for maintaining the reliability and robustness of our application.
    - Maintainability: Node-based coverage provides a clear, measurable metric that can be tracked over time, allowing us to monitor the quality of our testing efforts and identify areas of our application that require additional attention. This approach assists in pinpointing issues during development and future updates.
    - Holistic Testing: Node coverage also complements other testing criteria, such as branch coverage or path coverage. While branch coverage assesses the logical flow of our code, node coverage forms a fundamental foundation. By starting with node coverage, we ensure that each path or branch has been thoroughly examined, creating a more comprehensive testing strategy.

  
Figure 1: Decision tree for .add() function.

* Exploratory Testing
  + We identified a tool that would be as meaningful as possible to the code. We found that the documentation was helpful in describing the purpose of functions as well as the return type. Since most users will read the documentation to determine which function is needed for a given problem, we wanted to verify that it was as accurate as possible. This led us to using the Guidebook tool while using this technique.

**4.4 Test Categories:**

The goal of this test plan was to verify the functionality of the application. This is a light-weight date manipulation library, so we there were not any performance concerns. The library has no outside connections and not many dependencies, so there were also no major security concerns. Therefore, our test cases focused:

* Functionality
  + Goal: Verify software worked as expected. The test plan’s primary focus is on the functionality of the application. Given that it is a lightweight date manipulation library, the emphasis is on ensuring that all intended features and time-related functions perform as specified. This involves testing the core functionalities such as timezone information retrieval, movement between timezones, and other date object manipulations to guarantee their accuracy and reliability. It was important to consider comprehensive test coverage for each feature, including positive and negative test cases, edge cases, and scenarios that represent typical user interactions.
* Reliability
  + Goal: Verify the software is robust enough to handle bad inputs without crashing or causing errors. Reliability testing focuses on assessing the application's ability to handle unexpected or invalid inputs gracefully. This involves subjecting the software to various scenarios with incorrect or unusual inputs to ensure it does not crash and maintains stable performance. The goal is to enhance the application's resilience, providing a robust user experience even in the face of challenging input conditions. Reliability testing is crucial for identifying potential points of failure and improving the application's error-handling mechanisms. It involves intentionally introducing scenarios that might cause issues, such as incorrect data formats or out-of-range values, to validate the software's ability to gracefully handle such situations.
* Security
  + Not applicable: The application has no outside connections and not many dependencies, indicating that it operates in a relatively closed environment. As a result, no major security concerns are outlined in the test plan. Security testing, which typically involves assessing vulnerabilities and ensuring data integrity, confidentiality, and availability, may not be a primary focus in this context. While the application may not have significant security concerns, it's always good practice to conduct a basic security review, especially if the application interacts with sensitive data or if future updates might introduce new dependencies or external connections. This ensures a proactive approach to security, even in a lightweight library setting.

#### **Section 5: Test Cases**

**5.1 Test Group and Subgroup Definition:**

This is where you will define how the test cases will be structured and organized. Define test groups and subgroups for organizing test cases. Specify the objective for each group

The tests were separated by testing technique. The tests were written for and grouped by input domain modeling, graph-based testing, and exploratory testing.

**5.2 Test Cases:**   
List all test cases. Or provide a link to the test cases. Ensure that the link you provide is accessible to the instructor.

The Test Cases can be found in these two documents:

1. [InputDomain.test.js](https://pennstateoffice365.sharepoint.com/:u:/r/sites/SWENG581661/Shared%20Documents/Testing%20Report/inputDomain.test.js?csf=1&web=1&e=sqv3Wt)
2. [Exploratory\_Test\_Tour.pdf](https://pennstateoffice365.sharepoint.com/:b:/r/sites/SWENG581661/Shared%20Documents/Testing%20Report/Exploratory_Test_Tour.pdf?csf=1&web=1&e=nEAziM)

The provided test cases are part of a testing initiative for the "spacetime.how" application. The test plan is organized into various sections, and Section 5.2 focuses on detailing the test cases employed for verification.

The test cases encompass a wide array of functionalities, including time-related manipulations, timezone handling, date positioning, and various utility methods. The test plan primarily emphasizes the verification of functionality, ensuring that the application performs as expected. The application is described as a lightweight date manipulation library, suggesting a focus on correctness rather than performance optimization.

The detailed test cases are presented in the Test Anything Protocol (TAP) format, indicating whether each test case passes or fails. The majority of the test cases pass, confirming the expected behavior of the application. Notably, one test case labeled (3, century) fails due to a discrepancy between the expected and actual values.

The individual test cases span diverse scenarios, including timezone manipulation, date positioning, and seasonal computations. Methods for checking week numbers, quarters, and seasons, as well as handling month names, are thoroughly tested. Negative scenarios, such as handling null or undefined values, are also included to ensure comprehensive test coverage.

The test cases can be found in our automated test case file below or in the document describing the results.

**5.3 Traceability Matrix:**

To establish a clear linkage between requirements and corresponding test cases, a traceability matrix has been created. Please refer to the dedicated document titled "[Traceability Matrix](https://pennstateoffice365.sharepoint.com/:x:/r/sites/SWENG581661/Shared%20Documents/Testing%20Report/Traceabililty%20Matrix.xlsx?d=wcc2eff1be94b49baa49d3d566bddec33&csf=1&web=1&e=YN3qaK)" for a comprehensive mapping that outlines how each specific requirement aligns with its corresponding test case.

#### **Section 6: Test Environment**

**6.1 Multiple Test Environments:** List the different test environments used if applicable.

* Windows
  + GIT Bash – Node v21.1.0
  + Native windows terminal – Node v18
* Linux

**6.2 Schematic Diagram:**

Not Applicable

**6.3 Test Architecture Overview:**

The codebase is very modular, so we focused on nailing down solid unit test coverage on each individual component. Since everything is so well decoupled, there wasn't really a need for end-to-end test suites or complex integration testing. We were able to validate each isolated unit on its own merits without having to build elaborate flows. This let us laser focus the testing on the low-level functionality. Given the segmented structure of the code, that was the most efficient way to poke at each moving piece thoroughly.

**6.4 Equipment Table:**

* System Specs
  + OS: Windows 11
  + CPU: 12x Intel(R) Core i7-8700K @ 3.70 Hz
  + GPU: NVIDIA GeForce RTX 2080
  + RAM: 16GB

#### **Section 7: Testing Results:**

Overall, the code base is robust, as evidenced by the successful test result. The results of the test cases can be found in the following documents:

* [automated\_test\_results.txt](https://pennstateoffice365.sharepoint.com/:t:/r/sites/SWENG581661/Shared%20Documents/Testing%20Report/automated_test_results.txt?csf=1&web=1&e=0jWb4h) - This document contains the results from all the automated tests that were run. All tests passed successfully, indicating solid code quality.
* [Exploratory\_Test\_Tour.pdf](https://pennstateoffice365.sharepoint.com/:b:/r/sites/SWENG581661/Shared%20Documents/Testing%20Report/Exploratory_Test_Tour.pdf?csf=1&web=1&e=LnUkwO) - This document summarizes the exploratory testing that was performed. Some minor calculations issues were identified, but no major functions bugs were found.
* [Testing Report](https://pennstateoffice365.sharepoint.com/:f:/r/sites/SWENG581661/Shared%20Documents/Testing%20Report?csf=1&web=1&e=5E9QTI) - This directory contains all testing documentation, including graph, details test plans, test cases, and path coverage. It provides a comprehensive overview of the rigorous testing process that was followed.

#### **Section 8: Recommendations on Software Quality**

Our testing validated that the open-source codebase was thoroughly tested, with few defects uncovered. One area for improvement is fixing bugs in the .add() method when specifying multiple century values, as this was the only functional issue noted. We also saw some problems when incorrect data types were passed into methods. Adding TypeScript type checking could easily prevent these by enforcing type agreements. While annoying, these type problems did not actually break anything. Overall, open-source projects live and die by community testing, so the robustness seen here is a testament to those efforts. In additional detail - our extensive testing demonstrated the solidity of the code, with comprehensive examination showing strong performance. The sole functional bug was with .add() handling multiple centuries. Also, some errors popped up when wrong parameter types were used. TypeScript could lock down types and prevent this. These issues didn't impact the core library functionality though. In general, open-source quality relies on collaborative testing efforts to find and squash bugs.