The Impact of Test Case Clustering on Comprehending Automatically Generated Test Suites

Version of September 24, 2023

Longfei Lin

The Impact of Test Case Clustering on Comprehending Automatically Generated Test Suites

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Longfei Lin
born in Xinjiang, China

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

The Impact of Test Case Clustering on Comprehending Automatically Generated Test Suites

Author: Longfei Lin
Student id: 5483913

Abstract

Software testing, a critical phase in the software development lifecycle, is often hindered by the time-intensive and costly manual creation of test cases. While automating test case generation could mitigate these challenges, its adoption in the industry has been limited due to difficulties in comprehending the generated test cases. To address this, our study presents an approach for clustering test cases and evaluates its impact on the comprehensibility of test suites through empirical research. Our approach clusters test cases based on their covered objectives, grouping together those with similar attributes to enhance developer understanding. The core of our empirical research evaluates developer agreement with our clustering method and contrasts the comprehensibility of clustered versus non-clustered test suites. Findings suggest a broad agreement among developers in favor of our clustering approach, with clustered test suites facilitating faster software maintenance tasks. Notably, the effectiveness of task completion remained comparable between both suite types. In summary, our research introduces and validates an innovative test case clustering strategy, striving to enhance the comprehensibility of automatically generated test suites.

Thesis Committee:

Chair:	Dr. A. Panichella, Faculty EEMCS, TU Delft
University supervisor:	Dr. A. Panichella, Faculty EEMCS, TU Delft
Daily Co-Supervisor:	Ir. M. Olsthoorn, Faculty EEMCS, TU Delft
Committee Member:	Dr. W.P. Brinkman, Faculty EEMCS, TU Delft

Preface

As I write this preface, I realize it could potentially be the final piece required for my thesis, representing the culmination of my journey at TU Delft. My feelings at this moment go beyond joy, I am filled with deep gratitude for all those who supported and encouraged me throughout my thesis work. Today's achievements would have been unattainable without their support.

Firstly, I want to express my sincere appreciation to my supervisor, Annibale Panichella, for his kindness, continuous review of my work, and constructive feedback provided during our discussions. The realization of this research would have been impossible without his rigorous guidance. I also thank Mitchell Olsthoorn for being always available to answer questions, considering and discussing my propositions, and sharing his extensive experiences in academic exploration. His insights and encouragement have been crucial in motivating me to consistently strive for excellence. I also want to thank Willem-Paul Brinkman for agreeing to be part of my thesis committee and reviewing my work.

Finally, my deepest thanks go to my family and friends, and in particular, my parents, whose unwavering support has been the foundation of completing this journey. Your continuous encouragement, coupled with care, support, and understanding, has been my rock.

I hope this thesis serves as some form of inspiration to you. Enjoy the read!

Longfei Lin
Delft, the Netherlands
September 24, 2023

Contents

List of Figures

Chapter 1

Introduction

Software testing is one of the most crucial tasks in the software development lifecycle[9]. Developers are typically required to write unit test cases for their code implementation to ensure external quality during product delivery. Additionally, in some classical software development practices, such as eXtreme Programming (XP)[7], test cases are also viewed as a form of documentation, as they contain typical use cases, inputs, and exceptions for production code. However, software testing is also considered the most expensive task in the software development lifecycle, as the evolution of software systems necessitates frequent updates to test cases. Thus, traditional software testing tasks are deemed monotonous and time-consuming.

1.1 Problem statement

In recent years, a number of techniques have been proposed to automate the generation of test cases, with the aim of reducing the cost and effort associated with manual test case development. Among these, search-based approaches have emerged as a popular and effective option[31]. Such methods generate unit tests for a designated class under test (CUT) and seek to maximize structural coverage criteria for the CUT[27]. Although some studies have demonstrated that automated test case generation can be as effective as manual test case development in detecting faults[11], adopting automated test case generation tools within the industry remains limited. This is primarily due to the fact that, from the perspective of developers, the generated test cases still necessitate a significant amount of human intervention. Developers are required to invest considerable time in understanding and validating whether the generated test cases conform to their initial expectations[11, 30]. Furthermore, these test cases are often more difficult to comprehend than manually developed ones. Consequently, enhancing the comprehensibility of the generated test cases can significantly reduce the cost associated with software testing activities when using automatic test case generation tools.

Program comprehension is an extensively researched area, with existing empirical studies primarily focusing on the comprehensibility of source code[46]. These studies have made significant contributions toward comprehending source code from various perspectives, such as code readability, complexity metrics, and code visualization features. In contrast, research on the comprehension of test code has received less attention, which may be attributed to the comparatively simple structure and logic of test code relative to source code. However, some researchers have proposed techniques to improve the comprehension of test code[14, 26, 59]. Additionally, with the rise in popularity of automated test case generation in software testing, some researchers have examined the comprehensibility of automatically generated test cases[15, 16, 44]. Despite this, our investigation has revealed that almost all studies on test code comprehension, whether it is handwritten or automatically generated, have exclusively focused on the snippet-level, that is, the comprehensibility of individual test cases or test methods in JAVA and JAVASCRIPT, with no investigations conducted on the comprehension at the test suites level.

Consequently, to fill this gap, this study endeavors to develop a post-processing/ optimization approach for automatically generated test suites, aiming to diminish the efforts required by developers to comprehend such suites. Our approach tries to cluster multiple test cases with similar characteristics or behaviors into the same group and conducts readability improvement for individual test cases to enhance the comprehensibility of the whole test suite, which helps alleviate the developers' load of comprehending automatically generated test suites. To accomplish our research target, this study proposes a new test case clustering method based on the search objectives. We measure the similarity between test cases by examining the search objectives each test case covers, leading to their subsequent clustering. In this context, search objectives denote both function and branch objectives used as guides during meta-heuristic searches in the test case generation process. Due to the lack of a universally accepted standard for clustering test cases, an empirical study is essential to evaluate the effectiveness of our proposed method.

1.2 Research question

The empirical study principally seeks to address the following two research questions:

RQ1 *To what extent do developers agree with our approach of clustering the test cases?*

RQ2 *To what extent does test case clustering impact the comprehensibility of the generated test suites?*

We have designed three distinct software maintenance tasks based on the concept of learning activities related to program comprehension[38], each task evaluates the test case clustering methodology proposed in this study from a unique perspective. In our study, we used an empirical approach involving participants with varying professional backgrounds and expertise. Each participant completed three software maintenance tasks we designed, administered through an online survey. Their performance was evaluated to infer the impact of test case clustering on comprehensibility. Our results suggest that, in most instances, developers agree with the test case clustering approach we introduced. However, there were divergent opinions among developers for specific test cases. Concerning comprehensibility, we found that test suites using test case clustering method allowed developers to complete software maintenance tasks more quickly than those without clustering. Yet, when it came to the quality or effectiveness of task completion, no significant difference was observed in developers' performance between the two suite types.

In conclusion, the primary contributions of this research include:

- The development and validation of a novel test case clustering method specifically designed for automatically generated test suites.
- An empirical study evaluating the impact of test case clustering on the comprehensibility of such test suites.

1.3 Thesis outline

The remainder of this report is structured as follows. First, Chapter 2 provides background information and discusses related work relevant to this study. Chapter 3 details our methodology for clustering test cases. Chapter 4 describes the experimental design in-depth. In Chapter 5, we present the results of our empirical study. Chapter 6 discusses the key findings and outlines potential directions for future research. Finally, Chapter 7 concludes the paper with a summary of our contributions.

Chapter 2

Background and Related Work

This chapter presents the background and related work relevant to this study. We first review the automated test case generation and subsequently discuss the recent research on program comprehension.

2.1 Automated Test Case Generation

Automated test case generation is an ongoing research area that continues to captivate the interest of software testing professionals. With the growing size and complexity of software systems, the effort and expense of manually crafting test cases escalate. Automated test case generation can enhance the effectiveness and quality of software testing, diminish the costs and burden of manual testing[5], and aid developers in quickly and comprehensively detecting program bugs[11, 23]. There are two primary methodologies underpinning current research in this domain. The first, known as ”Static or Dynamic Analysis-based Approaches,” relies on analyzing the program’s code or behavior. The second, the ”Machine Learning-based Approaches,” leverages algorithms and data-driven techniques to predict and generate test cases.

2.1.1 Static or Dynamic Analysis-based Approaches

In recent years, the majority of research on automated test case generation has been grounded in static or dynamic analysis. These methodologies leverage the control or data flow graph of the software under test to generate tests that maximize coverage criteria. A plethora of automated techniques for test case generation has been proposed by researchers. These techniques can be broadly classified into the following categories: *random testing*[39], [12], *dynamic symbolic execution*[6], and *search-based software testing* (SBST)[37].

Random testing entails generating test cases based on the source code, done in an random manner. Its primary advantage lies in its simplicity and ease of implementation. However, the unit tests produced by this method can often be unclear and may result in lower coverage.

On the other hand, dynamic symbolic execution operates by gathering symbolic constraints from its execution path, formulating a specific formula. This formula is then solved using tools like SMT or SAT solvers, with the goal of covering various execution paths of the software under test. Despite the promising potential of dynamic symbolic execution in theory, practical applications encounter challenges. Path explosion and complexity constrains can make it difficult to apply to large-scale, real-world software systems.

Therefore, SBST has attracted growing attention from researchers. By defining suitable objective functions and search algorithms to cover different branches and paths of the program, SBST searches the test case space and generates runnable test cases. In comparison to random testing and symbolic execution, SBST frequently attains greater structural coverage percentage and can be applied at various levels of testing[37].

EVOSUITE[21], a prominent search-based software testing (SBST) tool, is the most popularly adopted framework for generating JAVA unit test suites automatically. Its diverse search-based test generation strategies include the WSA[22], MOSA[40], and DynaMOSA[41]. Among all these algorithms, DynaMOSA surpasses other genetic algorithms and demonstrates greater effectiveness and efficiency.[41].

Dynamic programming languages like JAVASCRIPT and PYTHON have gained significant traction in recent years. Consequently, researchers have shown increasing interest in developing automated test case generation tools based on dynamic typing, such as SYNTEST-JAVASCRIPT[50] and PYNGUIN[34]. However, when compared to statically-typed programming languages, dynamic languages pose an additional challenge: the handling of type information. Consequently, substantial efforts in research have been directed towards achieving type inference for variables. SYNTEST-JAVASCRIPT employs an unsupervised probabilistic type inference approach, striving for high coverage of the class under test. PYNGUIN utilizes type annotations present in the source code to deduce variable types.

SBST tools, despite producing tests with notable code coverage, often generate tests that lack the readability and comprehensibility of their manually written counterparts. This limitation stems from several factors: they frequently include non-descriptive variable names, random test inputs, and complex but confused assertions. As a result, developers face the added task—and associated costs—of rigorously reviewing these automatically generated tests.

2.1.2 Machine Learning-based Approaches

Since the rise of Large Language Models (LLMs) and their successes in Natural Language Processing (NLP), a growing number of researchers have been exploring their potential in the field of software engineering. Initially, LLMs are trained on vast datasets comprised of both natural language text and source code. Following this training, they can be fine-tuned for various software-related tasks, including test case generation. Here we provide a succinct overview of the machine learning-based test case generation approaches that have been proposed to date.

Siddiq et al.[49] conducted on a study to evaluate the capability of various LLMs in generating unit tests for two JAVA benchmark dataset. In their approach, they utilized three LLMs CHATGPT 3.5, CODEX, and CODEGEN and further investigated the impact of the input prompts on the LLMs' effectiveness. Their results highlighted that LLMs lagged behind in coverage when compared to manually written tests and those produced by EVO-

SUITE. This shortfall was particularly evident in the SF110 dataset. Additionally, a significant issue emerged regarding the test cases' compilation rate, with only 2.7% to 21% of the generated unit tests for the SF110 dataset being compilable across the studied LLMs.

Tufano et al.[52] introduced ATHENATEST, a system that leverages a BART transformer trained on a vast corpus of focal methods and related test cases in JAVA. In evaluations against prominent test case generation tools, namely EVOSUITE, ATHENATEST showcased noteworthy performance using the *Defects4J* dataset. Notably, it surpassed EVOSUITE in coverage in a majority of scenarios. However, it is worth noting that the overall correctness of the generated test cases was limited, achieving only 16.21%. Despite this limitation, a significant number of developers found ATHENATEST's test cases to be more readable and intuitive than those produced by EVOSUITE.

Lemieux et al.[32] introduced CODAMOSA atop PYNGUIN, integrating Codex to enhance the performance of SBST. CODAMOSA addresses a notable shortcoming in SBST: its potential stagnation when targeting test cases that encapsulate core program logic. SBST traditionally operates by inducing mutations in program test cases and selecting those showcasing superior fitness. To circumvent its limitations, CODAMOSA harnesses the power of LLMs to generate test cases for methods that often remain under-covered. Once generated, these test cases are translated into SBST's encoded code format, paving the way for mutation and fitness evaluations during the search process. Evaluative measures on a Python dataset demonstrated CODAMOSA's edge, revealing it achieved notably higher coverage across a more expansive range of the

Schäfer et al.[47] ventured into the realm of automatic unit test generation for JAVASCRIPT, ultilizing the capabilities of Codex. Their innovation led to the creation of a system named TESTPILOT. One of its distinctive features is an adaptive technique: if an initially generated test fails, TESTPILOT takes the proactive step of prompting the model again, using both the failed test and its associated error message, in a bid to craft a new, more effective test. When put to the evaluation, TESTPILOT's performance was gauged across a broad spectrum of 25 npm packages. Results painted a mixed picture. On one hand, the tests generated boasted a commendable median statement coverage of 68.2%. On the flip side, there was room for improvement in test correctness, which was found to be at a median of 47.1%.

In wrapping up, the deployment of LLMs in the realm of automated test case generation holds considerable potential. A standout benefit is the enhanced readability and comprehensibility of the test cases, especially when juxtaposed with static and dynamic analysis-based alternatives. Yet, it would be remiss not to highlight certain challenges. Chief among them is the considerable computational resource drain associated with LLMs. Moreover, the compilation and correctness rates of the generated test cases still leave room for improvement.

2.1.3 Syntest-JavaScript

In our study, we opted to employ SYNTEST-JAVASCRIPT for the generation of the requisite test suites. SYNTEST-JAVASCRIPT is a plugin of the SYNTEST-FRAMEWORK, a foundational library that offers universal interfaces tailored for crafting search-based software testing tools[50]. A notable feature of the SYNTEST-FRAMEWORK is its incorporation of the cutting-edge generic search algorithm DynaMOSA[41]. This algorithm has previously demonstrated its prowess within EVOSUITE, consistently yielding test suites that maximize structural coverage for target classes.

The procedure to generate test suites via SYNTEST-JAVASCRIPT is multi-faceted:

1. *Static Analysis*: Initially, SYNTEST-JAVASCRIPT undertakes a static analysis, pinpointing coverage objectives within the target class, encompassing function and branch goals. Concurrently, it conducts a preliminary type inference for pertinent components.

2. *Encoding and Initial Population*: Post-analysis, the identified objectives are encoded, and a initial population is generated through random sampling.

3. *Population Update*: The ensuing step is the iterative refinement of this population, aiming for more expansive objective coverage. This is orchestrated by harnessing the search algorithm interfaces present in the SYNTEST-FRAMEWORK, with iterations persisting until the designated search budget reaches its limit.

4. *Dynamic Analysis*: As the search progresses, SYNTEST-JAVASCRIPT cooperates dynamic analysis. Insights, such as `TypeError` details collected from the execution results of the existing population, update the type probability map assigned to each element. This step ensures a heightened precision in type inference.

5. *Final Construction*: Concluding the search, SYNTEST-JAVASCRIPT taps into the Archive set, a product of the SYNTEST-FRAMEWORK, to generate assertions and build the ultimate test suite.

2.2 Studies of Program Comprehension

2.2.1 Source Code Comprehension

Recently, researchers have explored the process of source code comprehension from various perspectives. Despite the absence of a perfect research method that elucidates the program comprehension process of developers entirely, studies have explored multiple facets that may impact it.

In a study conducted by Fakhoury et al.[20], functional near-infrared spectroscopy (fNIRS) was utilized to investigate the impact of source code lexicon and readability on developers' cognitive load during software comprehension tasks. The findings indicated that the presence of linguistic antipatterns substantially heightened cognitive load. However, the effects of readability and structure on cognitive load remained inconclusive.

In an effort to measure the comprehensibility of a code snippet, Scalabrino et al.[45] utilized multiple metrics, including code-related, documentation-related, and developer-related metrics. However, despite their efforts, the empirical results indicated that no existing metrics were specifically tailored for assessing code snippet comprehensibility.

Hofmeister et al.[29] studied the impact of identifier name length on program comprehension among developers. They found that words aided an average of 19% faster comprehension speed when compared to letters and abbreviations. Additionally, they found no significant difference in speed between letters and abbreviations.

2.2.2 Test Code Comprehension

While there has been less research on test code comprehensibility in recent years compared to source code comprehensibility, some researchers have proposed approaches to aid developers in comprehending test code.

Zhang[59] directed their attention to the simplification of tests at the semantic level and presented an algorithm called SimpleTest, which simplifies test cases by substituting expressions or statements with shorter alternatives, while considering the dependency relationships between variables.

Cornelissen and colleagues[14] strived to improve the intelligibility of test cases by abstracting their internal behavior into scenario diagrams via dynamic analysis, which illustrates the operation of the test cases. This visualization methodology preserves the required level of intricacy while removing superfluous information, resulting in a more accurate depiction of the interactions between objects. They demonstrated, through a thorough case study, that these scenario diagrams can enhance the comprehension of test cases.

Greiler et al.[26] proposed an automated approach to support software maintenance tasks that employs dynamic analysis of test cases to evaluate the similarity between the execution traces of high-level end-to-end (ETE) tests and fine-grained unit tests. This approach empowers developers to identify and replace flawed code segments with more efficient and dependable code snippets.

In addition to the aforementioned investigations on the comprehensibility of manually written test cases, there are also researchers who concentrate on the comprehensibility of automatically generated test cases. Daka et al.[15] developed a domain-specific model of readability based on human assessments for unit test cases and integrated readability evaluation as a secondary fitness function in EvoSuite to generate test cases that are more understandable for developers. Furthermore, Daka et al.[16] introduced and assessed an approach for deriving descriptive method names for automatically generated test cases. The effectiveness of this method in improving the comprehensibility of test code was validated from the perspective of developers.

2.2.3 Program comprehension as a Learning Activity

Bloom's Taxonomy[8] serves as a model for categorizing educational goals or learning standards. It helps to define the learning stage at which a learner is workign for a specific subject. The taxonomy includes six levels of cognitive learning: knowledge, comprehension, application, analysis, synthesis, and evaluation.

Recognizing the broad applicability of Bloom's Taxonomy, Fuller et al.[25] adapted this cognitive model for computer science, specifically software development. They grouped the cognitive levels into two semi-independent dimensions: 'Producing' and 'Interpreting'.

’Producing’ includes none, application, and creation, while ’Interpreting’ includes remembering, understanding, analyzing, and evaluating. They also introduced several learning activities that describe the cognitive skills involved in software development. These activities extend beyond ’learning’ to also include ’acting on knowledge’.

Building on Fuller et al.’s work, Oliveira et al.[38] expanded the learning activities and applied them to in the context of program comprehension, the full activities as shown in the following table.

Finally, Oliveira et al.[38] used Fuller et al.’s two-dimensional model to represent their extended learning activities. They also summarized the activities used in previous program comprehension research and used a heatmap (Figure 2.1) to show how often these learning activities appeared in past studies.

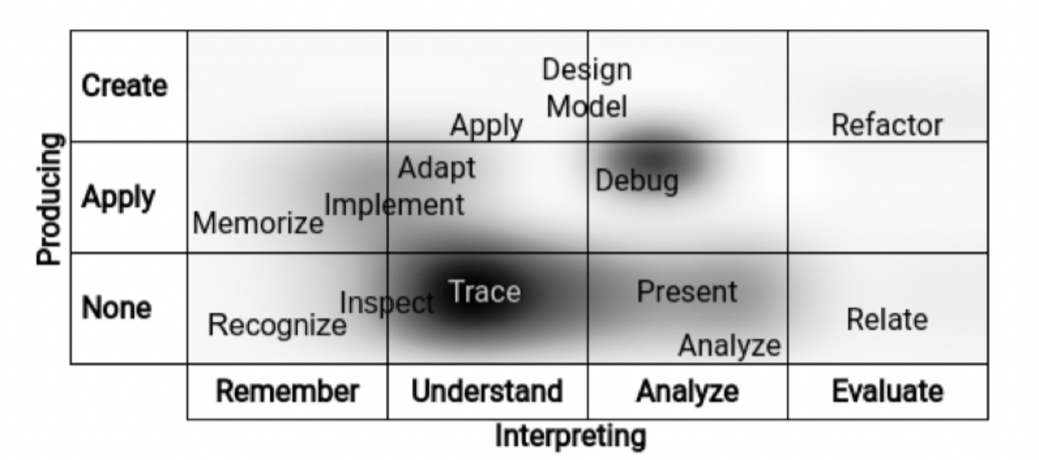


Figure 2.1: Two-dimensional model for learning activities and their frequency[38]

We aim to evaluate the impact of using test case clustering on the comprehensibility of automatically generated test suites. To do this, we need to design software maintenance tasks that require different cognitive abilities. We can then directly measure participant performance under two different test suites. This will allow us to indirectly draw conclusions about the influence of test case clustering on the comprehensibility of test suite. The two-dimensional model shown in Figure 2.1 helps us assess the cognitive difficulty and rationality of the tasks we design more easily.

2.2.4 Readability and Comprehensibility

In software engineering, the concepts of ”*readability*,” ”*understandability*,” and ”*comprehensibility*” play a crucial role in how developers interact with, interpret, and modify code. While these terms frequently surface in research discussions, they are often used interchangeably, leading to confusion. To provide clarity, we delineate and define each term distinctly:

Readability is fundamental to how effortlessly a developer can traverse through a piece of code or documentation. Building upon Buse and Weimer's[10] definition, we describe readability as "*a human judgment of how easy a text is to understand.*" It encompasses the accessibility of programming constructs, the coherence of coding idioms, and the clarity imparted by meaningful identifiers.

While understandability and comprehensibility might appear as distinct terms, we regard them as synonymous. Both pertain to the cognitive cost a developer undertakes to discern the intent behind a piece of code, grasp the connections among different code segments, and decipher the underlying semantics. To maintain consistency and avoid confusion, our subsequent discussions will favor the term "*comprehensibility*".

2.3 Clustering

2.3.1 Classic Clustering Algorithm

Clustering encompasses a collection of unsupervised learning algorithms that aim to classify data objects into distinct clusters. These clusters consist of objects that exhibit higher similarity to one another compared to objects in different clusters. Clustering methods are typically categorized into three groups: partitional, hierarchical, and density-based[35].

The renowned K-Means algorithm[17] is a partitional clustering method, partitioning data objects into non-overlapping clusters. For a dataset, it groups samples into k distinct clusters based on proximity. The intent is to maximize intra-cluster cohesion and ensure considerable inter-cluster separation. The process begins with the selection of initial k centroids. Determining the optimal number of clusters (i.e., k) and the initial cluster selection is crucial for model performance. As the initial centroids are often chosen randomly, different initialization methods can yield varied clustering outcomes. In subsequent steps, K-Means assigns data points to the nearest centroid using Euclidean distance metrics. The algorithm concludes by recalculating the centroid's position to the mean of its associated points.

Agglomerative Clustering[42], a hierarchical clustering method, adopts a bottom-up approach. Initially, each data point is treated as an individual cluster. These clusters are successively merged as one moves up the hierarchy, resulting in a dendritic structure visualized as a dendrogram. This dendrogram provides insights into the clustering stages and can be cut at different levels to derive the desired number of clusters.

DBSCAN (Density-Based Spatial Clustering of Applications with Noise)[19] stands as a representative of density-based clustering. It aims to group dense regions in feature space into clusters, while potentially leaving outlying points unassigned. DBSCAN has two main parameters: `min_samples` and `eps`. Here, `eps` represents a data point's radial neighborhood. When this neighborhood contains at least `min_samples` data points, that data point is deemed a core point. Clusters are then formed based on the proximity of these core points. Proper tuning of these parameters is essential for the algorithm's efficacy.

The three aforementioned clustering algorithms each have their distinct advantages and disadvantages. We encapsulate them in a Table 2.1:

Name of algorithm	Strengths	Weaknesses
K-means	1) Simplicity and fast. 2) Efficient implementations can deal with large datasets. 3) Easily interpretable results.	1) Pre-defined number of clusters. 2) Sensitive to initial places and outliers. 3) Assumes spherical clusters.
Agglomerative clustering	1) No require for number of clusters. 2) Work well with non-spherical clusters.	1) High computational complexity. 2) Not scalable.
DBSCAN	1) No require for number of clusters. 2) Handle arbitrarily shaped clusters. 3) Identify outliers.	1) Not suitable for clusters of varying densities. 2) Diffculty in determining parameters.

te

Table 2.1: Strengths and weaknesses of the clustering algorithms

2.3.2 Clustering Techniques for Software Testing

In software testing research, clustering techniques are mainly used for two primary targets: *Test Case Prioritization* and *Test Suite Minimization*.

Yoo et al.[57] proposed a clustering-based test case prioritization technique, where clustering depends on the similarity of faults that individual test cases detect, represented using dynamic execution traces. They used agglomerative hierarchical clustering, and the resulting dendrogram illustrates the cluster arrangement. By adjusting the threshold of this dendrogram, various cluster numbers can be derived. Fu et al.[24] presented a different approach, prioritizing test cases based on code coverage similarity. In their methodology, test cases with similar attributes are grouped into the same cluster using agglomerative hierarchical clustering. Empirical evidence supports the efficiency of their technique in improving fault detection rates.

Li et al.[33] undertook research on test cases for large-scale industrial applications. These test cases comprise sequences of testing steps written in natural language. To automate testing, developers first create a test method for each step, then write a script that invokes these methods in sequence. A challenge arises due to variations in natural language descriptions; developers sometimes miss steps that are semantically similar, leading to redundant test methods. To address this, Li et al.[33] proposed an approach that clusters similar testing steps. Their method, using domain-specific word embeddings and the Relaxed Word Mover's Distance metric, in combination with hierarchical agglomerative clustering and post-processing via K-means clustering, produced refined clustering results that can be manually adjusted.

Expanding on this, Viggiato et al.[54] aimed to reduce manual testing work and time by identifying redundant test cases. They utilized some most-recent sentence embedding models, including BERT and SBERT, to cluster similar testing steps described in natural language, then they could effectively identifies similar test cases by analyzing clustered testing steps.

Chetouane et al.[13] developed a technique that measures the Euclidean Distance between the inputs and outputs of different test cases. By coupling k-means clustering with binary search, their method effectively minimized the test case count without significant reductions in coverage or mutation scores.

On a similar note, Zalmanovici et al.[58] adopted a clustering approach to streamline the functional content analysis of expansive legacy test suites, thereby addressing the time and resource cost inherent to manual evaluations. Their technique measures distances rooted in textual similarities among test cases and leverages the DBScan algorithm for clustering purposes. Empirical results showed the approach's potential in substantially reducing the analysis time for the legacy test suites.

Contrary to the studies previously mentioned, our research focuses explicitly on test cases that are automatically generated, instead of those manually written. Further distinguishing our work, our objective for clustering test cases is not to address test case prioritization or test suite minimization problems, but rather to aid developers in gaining a better comprehension of the test suite.

2.3.3 Dimensionality Reduction Approaches

High-dimensional datasets pose significant challenges for effective clustering. As the dimensionality increases, the volume of the space expands exponentially, causing data to become sparse. In these vast spaces, individual data object tend to be distant from one another. This sparsity disrupts traditional notions of distance or similarity, rendering them less relevant. As a result, clustering becomes increasingly challenging—a phenomenon termed the "*Curse of Dimensionality*". Moreover, high dimensionality can amplify noise and heighten the risk of overfitting, further undermining clustering results.

Given these challenges, researchers often turn to dimensionality reduction techniques. These methods transform the high-dimensional data into a lower-dimensional space, making it more amenable to clustering algorithms. Dimensionality reduction techniques vary from linear methods, such as Principal Component Analysis (PCA)[56], to non-linear methods like t-distributed Stochastic Neighbor Embedding (t-SNE)[53], and even to deep learning-based approaches, namely autoencoders[28].

PCA operates by implementing an orthogonal transformation, converting a set of potentially correlated features into a linearly uncorrelated set. This transformation distills numerous features into a few composite features, termed as principal components. These components retain a significant chunk of the original data's information, with minimized redundancy. In essence, PCA achieves dimensionality reduction through a linear transformation.

Shifting from linear methods, the t-SNE algorithm seeks to preserve the local structure of data. It works by mapping data points onto a probabilistic distribution, using conditional probabilities to determine the similarity between them. The core idea is to ensure that the conditional probabilities in the high-dimensional space closely resemble those in the reduced space. This similarity-driven approach makes t-SNE especially adept for data visualization, emphasizing the preservation of data distinctiveness and local structures.

On the deep learning front, autoencoders emerge as a compelling dimensionality reduction technique. Designed to extract pivotal features, autoencoders can conduct both linear and non-linear transformations. Structurally, they comprise two main components: an encoder and a decoder. The encoder compresses high-dimensional data into a more compact representation, while the decoder's role is to use this representation to attempt a reconstruction of the original data.

Chapter 3

Approach

In this chapter, we primarily explain our approach to pre-process automatically generated test suites. Additionally, we will detail the specific algorithms and procedures we have undertaken for test case clustering. The overview structure of our approach is illustrated in Figure 3.1. The code implementing this approach is available in a public GitHub repository[1].

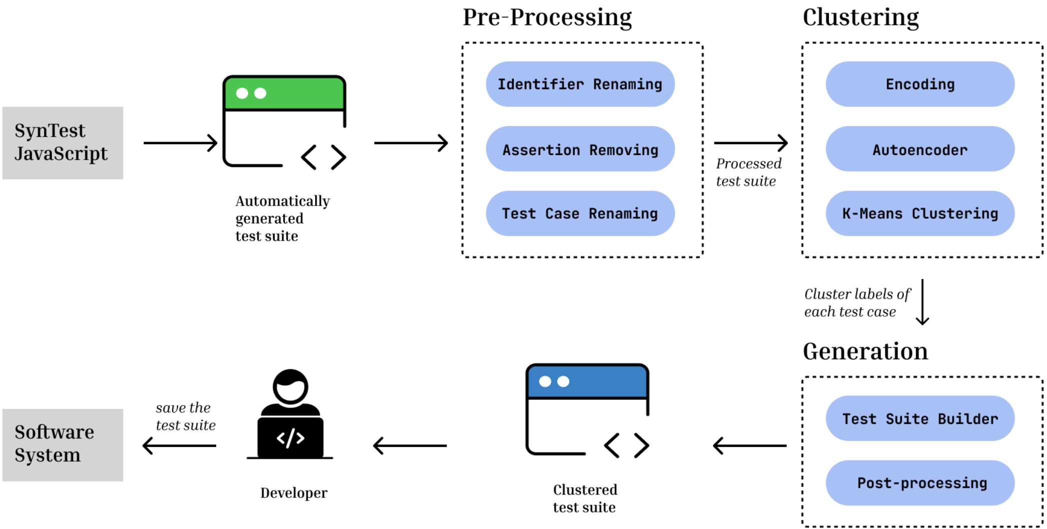


Figure 3.1: Overview structure of the approach

3.1 Test Suite Pre-processing

Before we conducting cluster algorithm on the automatically generated test suites, we first need to subject them to a series of pre-processing operations. These operations are

[1] https://github.com/LF-Lin/autogen-test-case-clustering

Listing 1 Example of an automatically generated test case using SynTest-JavaScript

```
it("test for Queue", async () => {
  const _Queue_object_sCRF = new Queue();
  const _data_object_IEwU = {};
  const _enqueue_function_VdcJ = await _Queue_object_sCRF.enqueue(
    _data_object_IEwU
  );
  const _toArray_function_RGyq = await _Queue_object_sCRF.toArray();

  expect(JSON.parse(JSON.stringify(_data_object_IEwU))).to.deep.equal({});
  expect(_enqueue_function_VdcJ).to.equal(1);
  expect(JSON.parse(JSON.stringify(_toArray_function_RGyq))).to.deep.equal([
    {},
  ]);
});
```

designed to enhance the readability of the test cases. As an example, Listing 1 shows an example of an automatically generated test case using SynTest-JavaScript. From this listing, it is clear that the variable names within the test case are relatively complex, including the types and unique IDs that were accumulated during the search process. In addition, the test case name lacks descriptiveness, and in the assertion section, there is an assertion that directly check the value for the variable `_data_object_IEwU`, making the purpose of the test case unclear. To address these readability issues, we have implemented the following pre-processing steps:

Listing 2 Test case after pro-processing

```
it("calls toArray after enqueue and returns array with length=1", async () => {
  const queue = new Queue();
  const data = {};
  const returnValue1 = await queue.enqueue(data);
  const returnValue2 = await queue.toArray();

  expect(returnValue1).to.equal(1);
  expect(JSON.parse(JSON.stringify(returnValue2))).to.deep.equal([{}]);
});
```

- *Identifier Renaming*. Renaming variable names within automatically generated test cases is a relatively straightforward task. These test cases primarily consist of three distinct types of variables: the object variable, which represents the instance of the class under test; variables used as parameters of the invoked methods; and variables for the returned values of method invocations.

  For the object variable, we simply replace its name with the name of the class under test and lowercase the first letter. For variables used as parameters for the invoked methods, we replace these variable names with the names of the method parameters in the source code of the class under test. This effectively enhances the readability of test cases because the source code of the class under test usually contains human-written variable names that are inherently readable. If duplicates occur among the replaced variable names, we append numerical indices to the names for distinction. For return values, we replace the variable names with `returnValue{index}`.

- *Assertion removing*. In automatically generated test cases, there are some assertions that are generated for variables of non-primitive types. Although these assertions do not cause the test case failure, they can create ambiguity about the purpose of the test case. To address this issue, we utilize the Abstract Syntax Tree (AST) to identify the variable definitions of non-primitive type and remove the corresponding assertions. The variable `_data_object_IEwU` in the Listing 1 is an example.
- *Test case renaming*. In Chapter 2, we introduced several different methods for test case renaming. After considering the effectiveness and implementation complexity of these methods, we chose to use Daka et al.'s coverage-based renaming method. This is the same test case naming methodology that is used in EvoSuite.

3.2 Test Case Encoding

In this study, we aim to use the objectives covered by the test cases, which incorporate both function and branch objectives, to characterize the unique attributes of the test case. Based on this, we implement our clustering algorithm. To compile the objectives each test case covers, we first need to implement source code instrumentation to facilitate tracking and logging of the coverage results of objectives for each test case. To ensure consistency between the coverage objectives we record and those used in the search process during test case generation, we use the instrumentation interface provided by SYNTEST-JAVASCRIPT to carry out source code instrumentation.

As a result, running the test suite on the instrumented class allows us to accurately determine which function objectives and branch objectives each test case hits during execution, as well as collect the conditions for the hit branch objectives.

Finally, we use binary encoding to encode the objectives covered by the test case. Each objective is represented by a binary digit, indicating whether the objective has been covered by the test case (1) or not (0).

3.3 Dimensionality Reduction using Autoencoder

Upon examining the data obtained after encoding the test cases, it becomes clear that the data is characterized by high dimensionality and sparsity. The dimensionality of the data is equivalent to the total number of function objectives and branch objectives in the class under test. As a single test case generally covers only a limited number of objectives, most of the values in the dataset are zero. This characteristic becomes more apparent as the complexity (number of branched) of the class under test increases, leading to the generation of more test cases to cover a larger number of objectives, rather than generating test cases that covered more objectives.

Moreover, data with high dimensionality and sparsity present challenges for clustering. In high-dimensional spaces, the distances between all data points tend to equalize, making distance-based clustering methods, such as K-means[17] or hierarchical clustering[42], ineffective. The sparsity of the data means that many dimensions may not contribute to the clustering results but still increase computational complexity. Additionally, sparsity can cause the distribution of data points in high-dimensional space to be highly dispersed, making it difficult for density-based clustering algorithms, such as DBSCAN[19], to identify sufficiently dense regions to form clusters. Furthermore, in high-dimensional data, the influence of noise and outliers can be magnified, potentially impacting distance-based clustering methods.

Given these challenges, it is crucial to perform dimensionality reduction on the encoded test case data.

In Chapter 2, various commonly used techniques for reducing the dimensionality of data were introduced. Given the characteristics of our specific test case data, we opted to employ an Autoencoder model in order to acquire a low-dimensional representation of the high-dimensional sparse data. This representation facilitates the processing of the data by the clustering algorithm.

An Autoencoder model comprises an encoder and a decoder. The encoder compresses the input data into a lower-dimensional representation, while the decoder reconstructs the original data using this low-dimensional representation. By minimizing reconstruction error, the Autoencoder model can identify and learn the significant features of the data, the neural newwork structure of Autoencoder is shown in Figure 3.2.

Following the construction of both the encoder and decoder, we proceed to train our model using our test case data. Once training is complete, we solely utilize the encoder to convert the original test case data into a low-dimensional representation. This representation effectively captures essential features of the data while disregarding noise and non-essential details.

3.4 Clustering using K-Means

After obtaining the features of the test case data through Autoencoder model, we apply K-Means algorithm to implement the test case clustering.

As the number of clusters needs to be set in advance when using K-Means, we determine the optimal number of clusters using the Elbow Method[51] and the Silhouette Coefficient[43].

The Elbow Method involves observing the relationship between the number of clusters and the Within-Cluster Sum of Squares (WCSS). As the number of clusters increases, WCSS gradually decreases. However, after a certain point, the rate of decrease in WCSS significantly slows down. This point, which looks like an ”elbow”, is called the ”elbow point”. The number of clusters corresponding to the ”elbow point” is considered optimal. However, the Elbow Method does not always clearly determine the optimal number of clusters, as there isn’t always an apparent ”elbow point”.

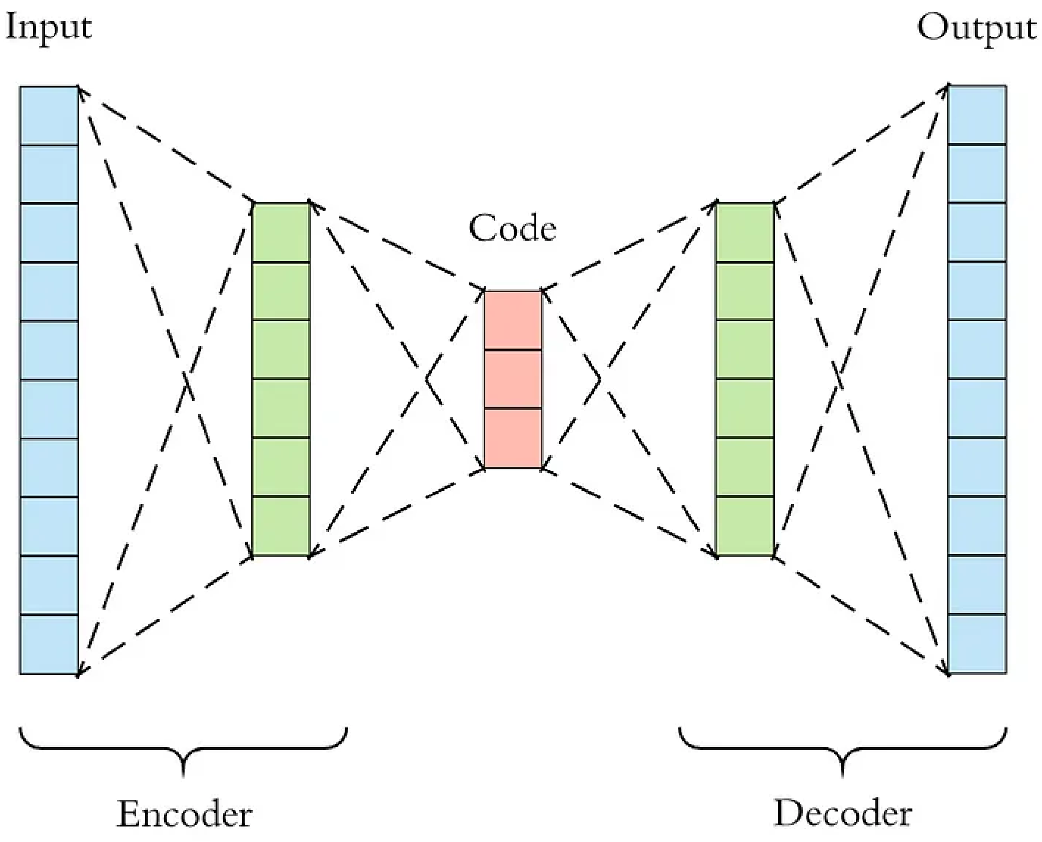


Figure 3.2: General architectual of the Autoencoder model[4]

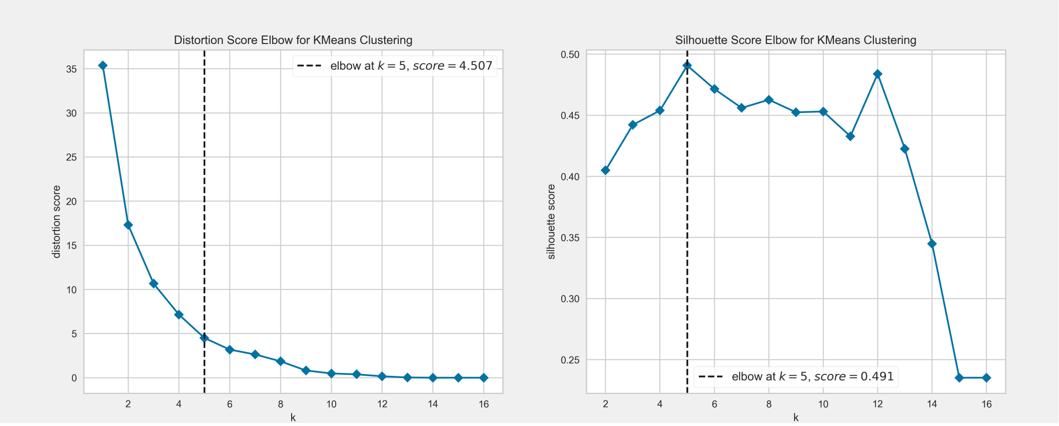


Figure 3.3: Example of using Elbow Method with the Silhouette Coefficient to determine optimal K value

Therefore, we also consider using the Silhouette Coefficient, a measure of clustering quality. The Silhouette Coefficient considers the cohesion and separation of the clusters, i.e., the compactness within the clusters and the separation between the clusters. A value close to 1 indicates that the sample points are very close to the other points in their cluster and very far from the points in other clusters, which is a good clustering.

By combining the Elbow Method with the Silhouette Coefficient, we can determine the optimal number of clusters for the current test case data. An example of using the above two methods to derive the optimal K value is shown in Figure 3.3.

3.5 Test Suite Generation with Clustering

Once we have determined the results of the test case clustering, that is, each test case has been assigned a label which represents the cluster to which that test case belongs. We need to create a new test suite to arrange and display the test cases after clustering. Fortunately, the JAVASCRIPT test framework we use, Mocha[2], provides a highly flexible way to structure test cases. Mocha introduces a nested test case organization style similar to RSpec Behaviour Driven Development style[3], which can be represented by various keywords that indicate different hierarchical relationships. The outline of the test suite is shown in Listing 3.

Listing 3 Structure of a test suite with test case clustering

```
describe("ClassUnderTest", () => {
  context("cluster: 1", () => {
    it("test case 1", () => {
      // ...
    });

    it("test case 2", () => {
      // ...
    });
  });

  // ...clusters

  context("cluster: n", () => {
    it("test case 1", () => {
      // ...
    });

    it("test case 2", () => {
      // ...
    });
  });
});
```

The structure of this testing suite uses three distinctive keywords, `describe`, `context`, and `it`, to portray a hierarchical relationship of ”test suite → clusters → test cases under the same cluster". This allows users to quickly identify test cases with similar features.

Chapter 4

Empirical Evaluation

To conduct an empirical experiment within the software engineering context, we employed the methodology proposed by Wohlin et al.[55] to design and delineate the experiment. The goal of this study is to evaluate how test case clustering affect the comprehensibility of a test suite, specifically focusing on its effectiveness and efficiency in aiding some software maintenance tasks. The results of the experiments are interpreted regarding the developer's perspectives, i.e., a developer who wants to understand whether using automatically generated test suite can reduce the cost of the maintenance tasks. To this end, we have designed an empirical evaluation to answer the following research questions:

RQ1 *To what extent do developers agree with our approach of clustering the test cases?*

Our primary objective in posing **RQ1** is to evaluate the acceptability and practical effectiveness of our test case clustering approach and its representation. When it comes to clustering methodologies, developers have varied interpretations of what constitutes ”similar test cases.” While clustering based on coverage objectives seems logical, it is uncertain if this aligns with the majority of developers' practical requirements and cognitive habits. On the topic of clustering representation, while organizing test cases within automatically generated test suites might appear straightforward, modern JAVASCRIPT testing lacks a universally accepted testing framework to achieve the clusters. We believe Mocha's nested structure represents clustering structure well, but its suitability and widespread acceptance among developers remain to be assessed.

RQ2 *To what extent does test case clustering impact the comprehensibility of the generated test suites?*

The primary goal of clustering automatically generated test cases is to help developers more easily understand each test case's intent and assess if the test suite meets their needs, ultimately aiming for less maintenance effort. However, it remains unclear whether clustering has directly positive or negative impacts on test case comprehensibility. To address this, we have formulated **RQ2**.

4.1 Experiment Definition

4.1.1 Tasks

Comprehensibility, being an abstract concept, cannot be quantified through direct metrics. As pointed out in related work, previous researchers have considered program comprehension tasks as learning tasks[38]. They have categorized a variety of learning activities, each representing different cognitive skills required in the process of program comprehension. By combining one or more of these learning activities into a learning task, researchers can indirectly measure program comprehensibility based on participants' performance in the given learning task. Following this established methodology, we will also measure the comprehensibility of test suites in a similar manner.

We combined one or more learning activities to create distinct *software maintenance tasks* related to software testing. Each participant was assigned the following tasks:

- **Fix the failing test cases**. The primary learning activity in this task is ”debug”, i.e., detecting and correcting flaws in a design. Participants are presented with the change history of the source code and an automatically generated test suite for the unmodified old version of the source code. They must first identify potential failing test cases with the help of the change history of the source code. Then, they are required to correct these test cases to ensure the test suite remains valid for the modified new version of the source code.

- **Identify potential error scenarios**. The primary learning activities in this task are ”recognize” and ”inspect”. Participants are provided only with an automatically generated test suite, without access to the source code for the class under test. They must determine the input boundaries and types for a specified method by examining related test cases in the provided test suite, thereby identifying the conditions under which the method would throw an exception. Additionally, participants are required to rate their understanding of the class under test functionality and their confidence in that understanding.

- **Evaluate and cluster the test cases**. The main learning activity in this task is ”inspect” and ”design”. First, we presented participants with the results of our test case clustering approach and asked them to assess their level of agreement with each cluster. Then, we asked participants for their perspectives on the criteria or conditions they consider important for test cases to share common features and behaviors if they were responsible for the clustering approach.

4.1.2 Objects

Class under test

To construct the test cases required for the survey, we selected three JAVASCRIPT classes to serve as classes under test. These include: (1) `Polygon.js`, which implements a polygon

Class Under Test	#branch objective	#function objective	SLOC
`Polygon.js`	18	9	161
`Queue.js`	12	8	110
`ShoppingCart.js`	22	11	98

Table 4.1: Statistics of the classess under test

that can handle a variable number of vertices, (2) `Queue.js`, which implements a First-In-First-Out queue, and (3) `ShoppingCart.js`, which implements a shopping cart that can manage a variety of items. Each of these classes has more than 10 branches, making them suitable for coverage by test case generation tools, more information about the classes under test is shown in table 4.1.

Test suite generation

After selecting the classes under test, we used the tool SYNTEST-JAVASCRIPT to generate test suites for these classes. We used the default parameter configurations and fine-tuned DynaMOSA presets config to collect the raw test suites, then applied the preprocessing and clustering methods discussed in Section 3 to generate the test suites for the experiments.

It is important to note that SYNTEST-JAVASCRIPT is still under development, and the test suites it generates may not be easily readable. To improve the readability of the test suites for participants, we made the following assumptions to JAVASCRIPT:

1. For variables of the `number` type, it can generate either integer or floating-point data based on the context of the class under test.
2. Strings are composed of random identifier names from the source code, rather than completely random characters.

Change histories of the source code in Task 1

In Task 1, as outlined in the tasks, we provide participants with a code change history to help them identify potentially failing test cases caused by the changes. The code changes we made meet the following criteria:

1. All changes must be related to a branch. This means that each modification is associated with a specific branch of the code.
2. The code changes are independent of each other, meaning there is no interrelation between them.

4.1.3 Participants

We assume that people participating in the survey have basic experience with software testing and JAVASCRIPT testing frameworks. This study is open to developers from different

JAVASCRIPT experience	Count	Background	Count
<1 year	4	Software developer	29
1-2 years	15	Student	18
3-5 years	22	Researcher	5
6-10 years	9		
>10 years	2		

Table 4.2: Demographic data of participants

backgrounds, such as students, industry professionals, and researchers. Invitations are being distributed through email and social media platforms. In addition, we are collecting information about participants' experience with automatic unit test generation tools before the survey begins. Upon completion of the survey, all participants will receive equal compensation. To ensure the quality of survey responses, qualifying questions have been included to exclude participants who rush through the survey, as this may indicate that they are not seriously engaged with the survey questions. The JAVASCRIPT programming experience and background of the participants is presented in the table 4.2.

4.2 Hypotheses Formulation

For **RQ1**, we can formulate the null hypotheses to be tested as follows:

- H_1 Developers do not agree with the approach of clustering the test cases.

For **RQ2**, we can formulate the null hypotheses to be tested as follows,

- H_{2_1} There is no difference in the effectiveness of finishing the test code comprehension tasks between clustering and non-clustering test suites.
- H_{2_2} There is no difference in the efficiency of finishing the test code comprehension tasks between clustering and non-clustering test suites.

As explained in Section 4.1.1, comprehensibility is not a directly measurable metric. Therefore, we use two learning tasks as proxies for comprehensibility. With this approach, the main hypotheses can be broken down as follows:

The null hypotheses for Task 1:

- $H_{2_{11}}$ There is no difference in the effectiveness of fixing the failing test cases between clustering and non-clustering test suites.
- $H_{2_{12}}$ There is no difference in the efficiency of fixing the failing test cases between clustering and non-clustering test suites.

The null hypotheses for Task 2:

Dependent Variable	**Explanation**
Task 1	
Selected test cases	Number of test cases selected by participants that they thought might fail
Fixed test cases	Number of test cases correctly fixed by participants
Time spent	Total time spent on this task in minutes and seconds
Task 2	
Selected input conditions	Number of input conditions selected by participants that they thought might cause the method to throw an error
Time spent	Total time spent on this task in minutes and seconds
Task 3	
Level of agreement	Developers' opinions on our clustering approach

Table 4.3: Dependent variables

- $H2_{21}$ There is no difference in the effectiveness of identifying potential error scenarios between clustering and non-clustering test suites.

- $H2_{22}$ There is no difference in the efficiency of identifying potential error scenarios between clustering and non-clustering test suites.

However, it is important to note that the effectiveness of test case clustering can depend on several factors, including the complexity of the system under test, the quality of the test cases, and the clustering algorithm used. Therefore, all null hypothesis for **RQ2** are *two-tailed*, because there is no assumption regarding test suites with clustering being better than test suites without clustering.

4.3 Variables

4.3.1 Independent and dependent variables

In preparation for the experimental design, it is crucial to establish the independent and dependent variables that will be utilized.

For **RQ1**, we primarily conduct an observational study focused on the perception of a specific approach. Since this study lacks an independent variable, we consider the feedback or viewpoints of developers (their level of agreement) in Task3 as the dependent variable.

For **RQ2**, the main independent variables are the different test suites used in Task 1 and Task 2. The test suites are divided into two categories: (1) automatically generated test suites with test case clustering and (2) automatically generated test suites without test case clustering. The effects of these two different treatments will be evaluated by analyzing the dependent variables derived from the data collected in Task 1 and Task 2. Table 4.3 shows the dependent variables implemented in this experiment.

In Task 1, participants select test cases that may potentially fail and explain why these cases could lead to failure. By comparing the selected test cases to actual failures, we can evaluate participants' accuracy in identifying potential failure cases. Also, by analyzing the explanations given by participants, we can evaluate their accuracy in correcting potential failure cases. The total time spent by the participants on this task is also recorded. Thus, we can calculate the efficiency of test case correction by dividing the number of correctly fixed test cases by the total time spent.

In Task 2, participants are presented with a test suite and given possible types and values of input parameters for a particular method. They must then determine whether an exception could occur for that method. By comparing the participants' choices to the correct results, we can evaluate their accuracy in identifying exception conditions. Similar to Task 1, the total time spent on this task is recorded to calculate efficiency by dividing the number of correctly selected exceptions by the total time spent.

In Task 3, participants review the test case clustering results. Using a 5-point Likert scale, they provided feedback on whether they believed the test cases within a cluster shared common features or behaviors and should be grouped together.

In summary, we used Task1 and Task2 to address **RQ2**, while Task3 catered to **RQ1**. The mismatch in task order with the research questions was deliberate. Our goal was to mask specific research background and objectives from participants, reducing potential biases.

4.4 Design

In **RQ1**, due to the lack of an independent variable, we do not have a control group in experiment. As a result, all participants are exposed to the same content in Task 3.

In contrast, the design for **RQ2** is considerably more complex. For Task 1, we used a *within-subjects counterbalance design*[55] to ensure the validity and reliability of our evaluation of the impact of test case clustering. This experimental design was specifically chosen to control potential order effects.

In the survey, we used two different classes under test. Each class has two variants of the test suite: one with test case clustering and one without. We assigned the experimental tasks to participants in two different orders. The first group started with a clustered test suite for the first class under test, then moved on to a non-clustered test suite for the second class under test. The second group started with a non-clustered test suite for the first class, then moved on to a clustered test suite for the second class. This ensured that each participant experienced both levels of our independent variable.

This arrangement of treatments ensured that potential impacts caused by the sequence of tasks, such as familiarity or fatigue, were evenly distributed across both cases. By incorporating this balanced design into our research, we aimed to eliminate any potential bias resulting from the order of tasks, thereby ensuring a robust check on our results.

In Task 2, we use the same independent variable as in Task 1, but we employ a different experimental design: *completely randomized design*[55]. This is a common experimental design for comparing two treatments. In the survey, participants are randomly assigned a treatment, and there is only one class under test in this task for both treatments.

In Task 1, each participant undergoes all treatments, which provides multiple data points per participant. In contrast, in Task 2, each participant undergoes only one treatment, providing a single data point. Although the experimental design of Task 2 results in fewer data points, we believe it is still worthwhile for the following reasons:

Balancing Statistical Power and Practicality: Although the within-subject design of Task 1 provides more data points per participant, it also requires more time and resources from each participant. The completely randomized design in Task 2 might collect fewer data points per participant, but it is more manageable and potentially less taxing on the participants.

4.5 Procedure

This study was conducted using an online survey. A third-party platform, Alchemer[1], specifically designed for interactive surveys, was used to create the survey content.

Before starting the tasks, we required participants to complete a pre-task questionnaire. This allowed us to gather important information about their experience in software testing and JAVASCRIPT development. In addition, we provided a brief context related to software maintenance activities before each task to help participants understand the nature of the task. However, we do no provide any assumptions related to the study to the participants.

After completing the tasks, participants were asked to complete a post-task questionnaire for each task. These questionnaires contained a series of open-ended questions and rating questions on a 5-point Likert scale that allowed participants to evaluate task-related issues. These post-task questionnaires also helped us validate our study results.

4.6 Analysis Method

First, we use the *Shapiro-Wilk test*[48] to check if the data follows a normal distribution. If the p-value is less than 0.05, this indicates that the data can reject the null hypothesis of normal distribution. The results showed that for our data related to effectiveness, such as the number of correctly fixed test cases from task 1 and the number of correctly selected anomalous inputs from task 2, these data do not follow a normal distribution. However, for data related to efficiency, the p-values are more than 0.05. This means we cannot reject the null hypothesis, and these data do not significantly deviate from a normal distribution.

Although some data follow the normal distribution, non-parametric statistical test is still a better choice used to compare different types of results in our study. Because parametric tests, such as the commonly used independent samples t-test, require data to follow the normal distribution and assume equal variances between different treatments. On the other hand, non-parametric tests like the *Wilcoxon rank-sum test*[36] can be more robust and less affected by outliers. Therefore, we employ the Wilcoxon rank-sum test with a p-value threshold of 0.05 in this study. Furthermore, we utilize Cliff's Delta to measure the magnitude of difference between two treatments.

Chapter 5

Results

In this chapter, we report results of our empirical study, with the aim of answering the research questions formulated in Chapter 4. The experimental results and data analysis procedures can be accessed on the TU Delft Project Data Storage[1].

5.1 RQ1: To what extent do developers agree with our approach of clustering the test cases?

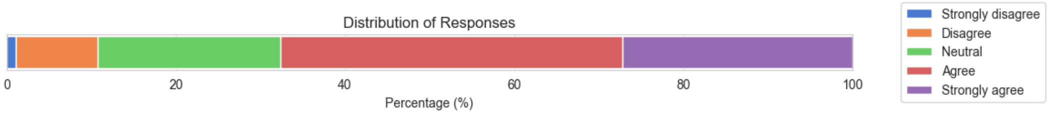


Figure 5.1: 5-point Likert agreement scales results for clustering approach

Figure 5.1 shows the results of the Likert agreement scales used for all clusters in Task 3. The results reveal that most participants agreed with our clustering outcomes. Specifically, 40.4% chose 'Agree' and 27.2% chose 'Strongly agree', showing strong endorsement. However, a significant portion of responses (21.7%) were neutral, neither agreeing nor disagreeing with our clustering results. Only a small fraction of participants (9.6%) disagreed.

Figure 5.2 clearly illustrates and compares the results of agreement inquiries about our classification approach for test case instances, as evaluated by participants. From the diagram, we can see that in most clusters, more individuals chose 'Agree' or 'Strongly agree' than 'Disagree' and 'Strongly Disagree'. This suggests that participants generally agree that the test cases in these clusters share common features, justifying their grouping. However, for 'Cluster3' and 'Cluster6', more individuals selected 'Neutral' or 'Disagree'. This shows a clear divergence between the participants' perspectives and our clustering results.

The first point of contention is within 'cluster6', where participants' disagreement was most noticeable. The code snippet of 'cluster6' is shown in figure 5.3. Participants' responses to 'cluster6' varied, with a majority choosing 'Disagree'. Figure 5.3 clearly shows

[1] `https://webdata.tudelft.nl/staff-umbrella/thesis01/data/`

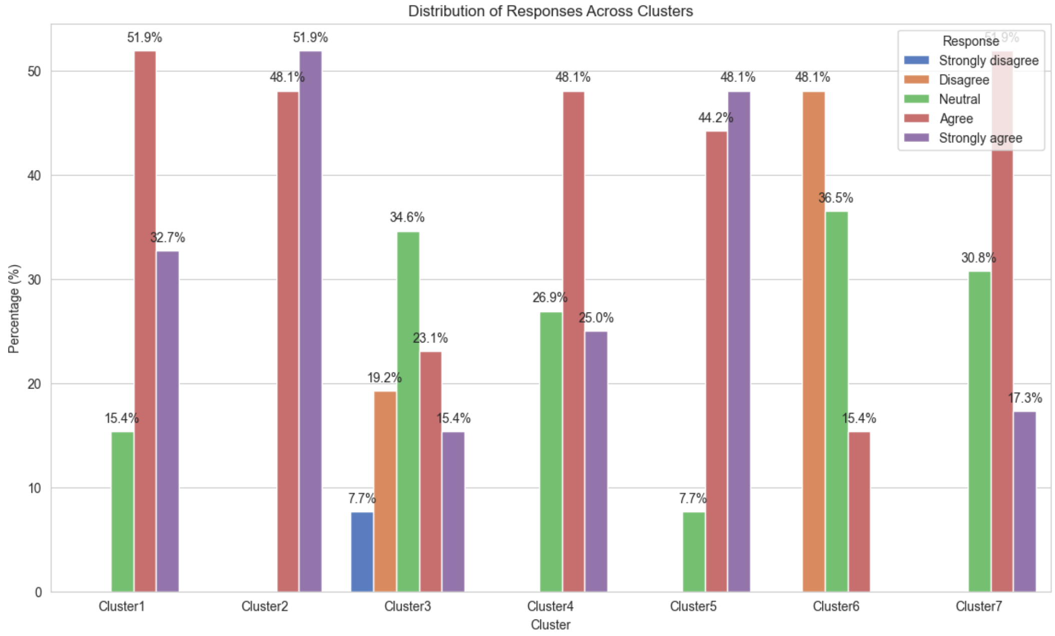


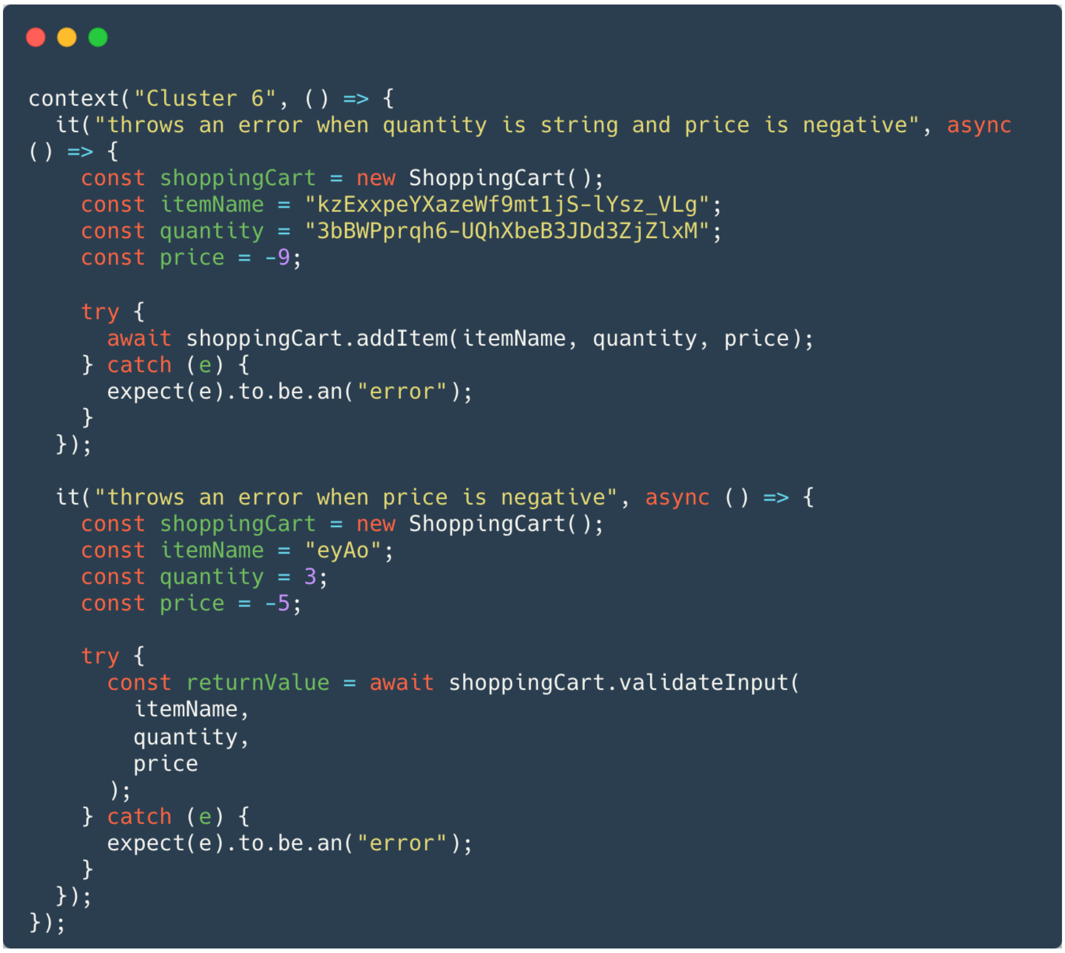
Figure 5.2: 5-point Likert agreement scales results for clustering approach across all clusters

that the main difference between the two test cases is the methods they invoke: the first test case uses `addItem`, while the second uses `validateInput`. As a result, many participants argued that these two instances don't share common features or behaviors.

However, our criteria of similar test cases is based on their covered objectives. In the source code of the class under test, `addItem` needs to call `validateInput` to check the type of its input parameters, then it executes the logic of adding items. Therefore, for these to test cases, both of them cover the function objective `validateInput`. Also, the variable `price` in both cases is a negative value, which means both test cases cover the true branch of the condition `typeof price !== "number" || price < 0` within the function `validateInput`. This is why we grouped these two test cases together.

A good example of consensus is 'cluster2', where the responses showed the least disagreement. All participant responses were 'Agree' or 'Strongly agree'. As we can see from Figure 5.4, the main actions of the two test cases in 'cluster2' are simply to invoke the `applyDiscount` method. However, one test case triggers the true branch of the conditional statement `typeof discount !== "number" || discount < 0 || discount > 1`, while the other triggers the false branch. This ensures that this cluster covers all branches of the `applyDiscount` method. Therefore, participants unanimously agreed with this cluster result.

To address **RQ1**, we applied statistical testing to challenge the null hypothesis discussed in Chapter 4. Given our data's ordinal nature, we used the one-sample Wilcoxon signed-rank test. We aimed to determine if the median level of agreement significantly differed from 3 (neutral). If the test showed a significant difference from neutral and the observed



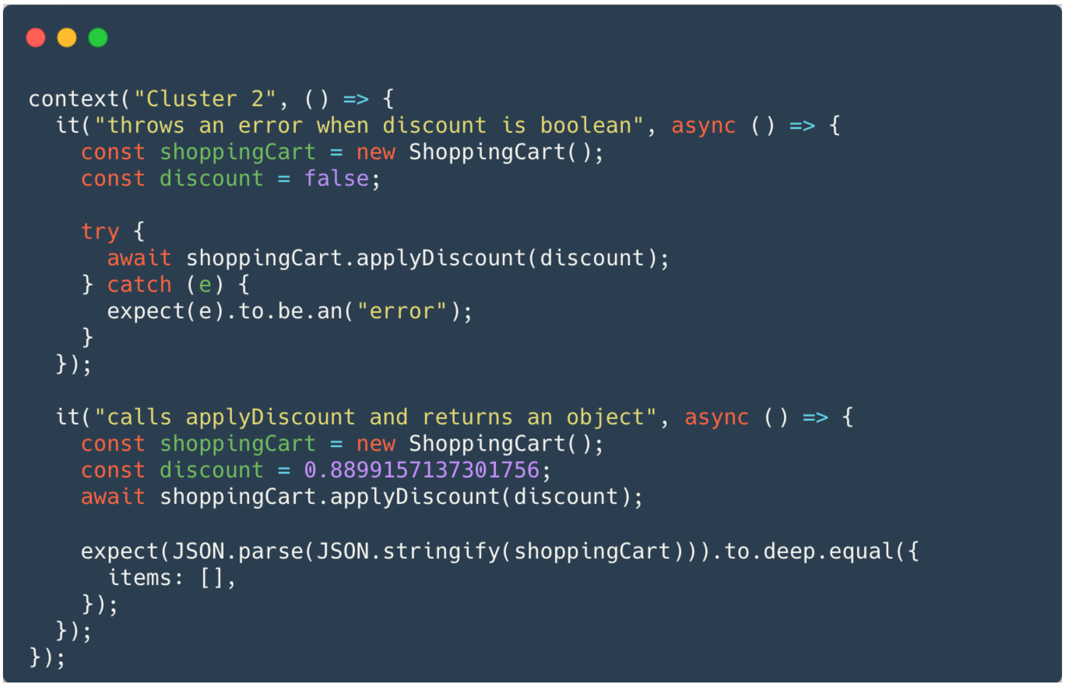
```
context("Cluster 6", () => {
  it("throws an error when quantity is string and price is negative", async
() => {
    const shoppingCart = new ShoppingCart();
    const itemName = "kzExxpeYXazeWf9mt1jS-lYsz_VLg";
    const quantity = "3bBWPprqh6-UQhXbeB3JDd3ZjZlxM";
    const price = -9;

    try {
      await shoppingCart.addItem(itemName, quantity, price);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error when price is negative", async () => {
    const shoppingCart = new ShoppingCart();
    const itemName = "eyAo";
    const quantity = 3;
    const price = -5;

    try {
      const returnValue = await shoppingCart.validateInput(
        itemName,
        quantity,
        price
      );
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });
});
```

Figure 5.3: Code snippet of ’cluster6’

median was above 3, it would imply a preference for our clustering approach among developers. Conversely, an observed median below 3 would indicate disagreement.

Our test results show robust evidence against the null hypothesis, suggesting a median of 3. With an observed median agreement level of 4 with $p << 0.05$, it is clear that developers generally agree with our clustering approach. When applying the Wilcoxon test individually to each cluster, most clusters (except ’cluster3’) showed a very low p-value (much less than 0.05). This suggests a significant difference from ”neutral” for these clusters. However, ’cluster3’ had a p-value above 0.05, indicating that its median agreement level does not significantly differ from ”neutral”.

In our qualitative analysis, we asked participants to explain their reasoning for selecting ”strongly disagree” or ”strongly agree” for each cluster. When participants chose ”strongly agree”, they often cited reasons such as *the test cases invoking the same method* or the *test cases being against similar inputs*. On the other hand, for ”strongly disagree”, common reasons included *test cases involving different methods*, *using different inputs*, or *having*



```
context("Cluster 2", () => {
  it("throws an error when discount is boolean", async () => {
    const shoppingCart = new ShoppingCart();
    const discount = false;

    try {
      await shoppingCart.applyDiscount(discount);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("calls applyDiscount and returns an object", async () => {
    const shoppingCart = new ShoppingCart();
    const discount = 0.8899157137301756;
    await shoppingCart.applyDiscount(discount);

    expect(JSON.parse(JSON.stringify(shoppingCart))).to.deep.equal({
      items: [],
    });
  });
});
```

Figure 5.4: Code snippet of 'cluster2'

divergent test purposes.

It is important to note that these cited reasons are the most common and do not capture every developer's viewpoint. For example, in 'cluster6' in figure 5.3, proponents felt that the test cases had identical input parameters and emphasized exception handling. In contrast, those in strong disagreement pointed out the distinct methods being invoked.

Post-task Questionnaire

As shown in Figure 5.4 and Figure 5.3, we used the `context` keyword from the Mocha testing framework to define the clustering level within the test suite. The whole clutering structure of a test suite is presented in Listing 3. While previous testing practices did not typically feature clustered test cases within JAVASCRIPT test suites, we believe this nested approach represents the clustering structure well. However, its practicality requires further validation from developers. To validate this idea, we asked: "*Do you agree that using the above test suite structure is a good way to organize/cluster the test cases in previous task?*" Participants' responses can be seen in Figure 5.5.

Most responses agree with the effectiveness of the nested 3-layer structure from Listing 3 for representing test case clustering results. Analyzing the response data using the One-sample Wilcoxon signed-rank test yielded a p-value of 0.045, indicating that the median level of agreement significantly differs from 3 (neutral). In other words, a majority of respondents agree that using the above test suite structure is a good way to represent the results of test case clustering. The significance of the Wilcoxon test result further confirms

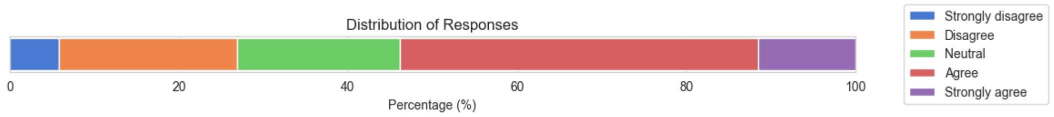


Figure 5.5: 5-point Likert agreement scales results for clustering structure of test suite

that this agreement is not by random chance and represents a genuine consensus among respondents.

> **RQ1**: Based on our statistical tests, we successfully rejected the null hypothesis H_1, indicating a general agreement among developers with our clustering approach. When we looked at individual clusters, only one did not show a significant difference from neutral. In summary, most participants agree with our results for the majority of the test case clustering, but they disagree with a few of the clustering outcomes.

5.2 RQ2: To what extent does test case clustering impact the comprehensibility of the generated test suites?

To address **RQ2**, which focuses on the influence of test case clustering on comprehensibility, we formulated two software maintenance tasks. In alignment with the null hypotheses outlined in Chapter 4, we have further subdivided **RQ2** into two sub-research questions, which are detailed in this section.

5.2.1 RQ2.1: To what extent does test case clustering impact the developer's ability on fixing the failing test cases?

Effectiveness of fixing the failing test cases

Figure 5.6 displays two violin plots which illustrate the effectiveness for two different treatment with two classes under test, namely `Polygon.js` and `Queue.js`. In Task 1, effectiveness is defined as the number of correctly fixing of test cases. Our findings reveal that when working on the test suite that generated from the `Polygon.js` class, participants were able to identify and fix a greater number of test cases, i.e., better effectiveness, by utilizing test case clustering. However, the Wilcoxon test indicates there is no significant difference between two treatments with a p-value of 0.125 and a medium effect size 0.246. Moreover, when participants working on the test suites generated from the `Queue.js` class, there is also no significant difference (p-value=0.765) in effectiveness between the test suites that employ test case clustering and those that do not, which means that our test case clustering will not help the participants to identify and fix more test cases.

Although the difference is not statistically significant, at the very least, the use of test case clustering in the test suite does not cause a deterioration in participant performance in this task.

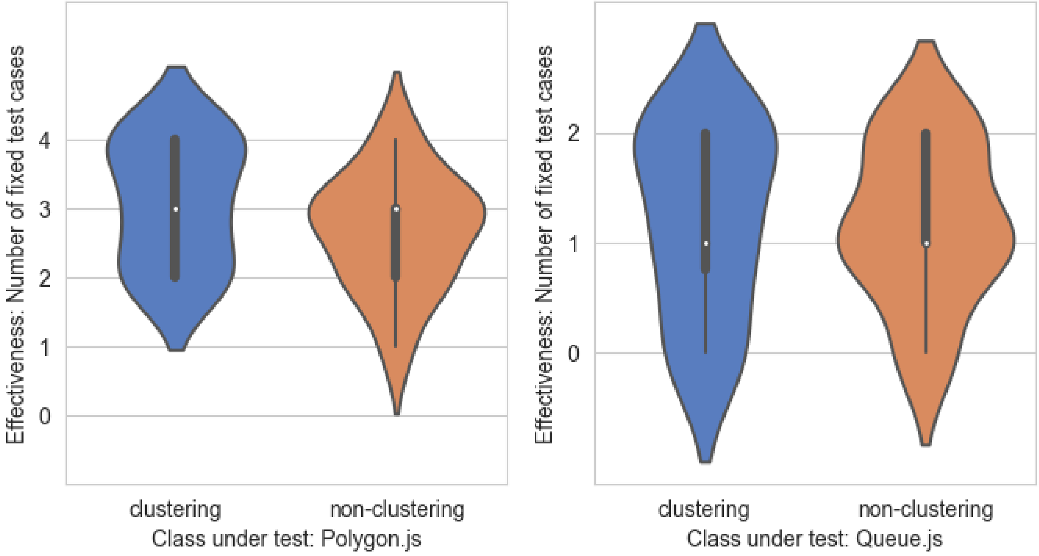


Figure 5.6: Violin plot for effectiveness results of task 1

Efficiency of fixing the failing test cases

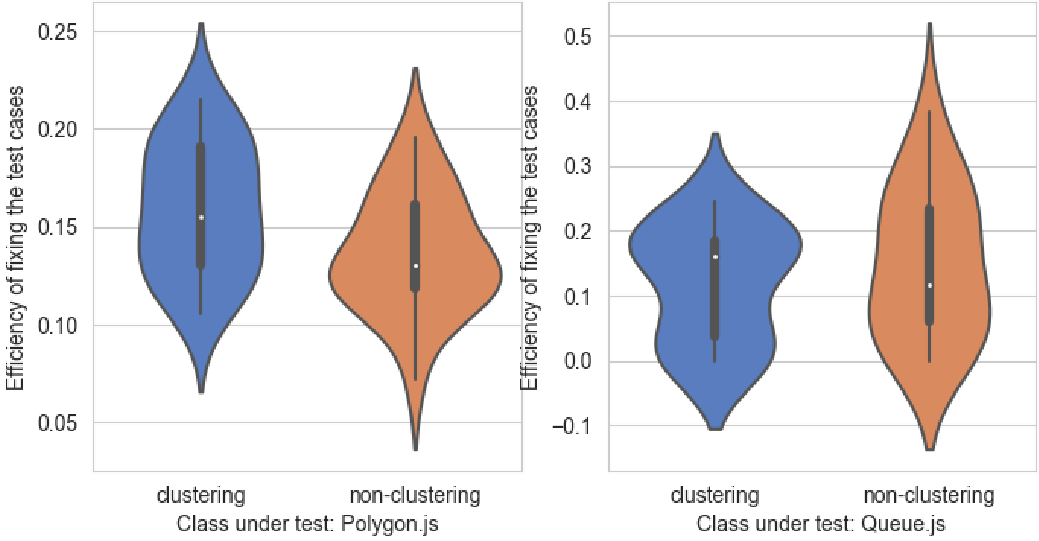


Figure 5.7: Violin plot for efficiency results of task 1

Figure 5.7 shows two violin plots that represent the efficiency results for treatments from two different classes under testing: `Polygon.js` and `Queue.js`. In Task 1, we define efficiency as the number of test cases that are correctly fixed divided by the time participants spend on the task. Consistent with previous results, when participants work on the test suites generated from the `Polygon.js` class, a test suite with test case clustering is significantly more efficient than a test suite without test case clustering. The Wilcoxon test gives a p-value of 0.0246 with a medium effect size of 0.380. However, for test suites generated from the `Queue.js` class, there is no significant difference (p-value=0.315) in efficiency between the different treatments.

One possible explanation for these results is that the effect of test case clustering is less noticeable in the test suite generated from the `Queue.js` class. Compared to `Polygon.js`, `Queue.js` has fewer cyclomatic complexities and branches. Therefore, when using SynTest-JavaScript to generate corresponding test cases, the number of test cases is also fewer (17 test cases vs 8 test cases). When participants read a test suite of this length, they can easily remember the differences and similarities between various test cases without being confused by excessive code. As a result, the advantages of test case clustering might not be able to show its full potential.

Post-task Questionnaire

After completing the task, we asked participants to optional fill out a post-task questionnaire to gather their subjective evaluations of the task. The questionnaire included a multiple-choice question where we asked participants to choose what they found most helpful in the test code while fixing test cases. The results of this question are shown in the following figure.

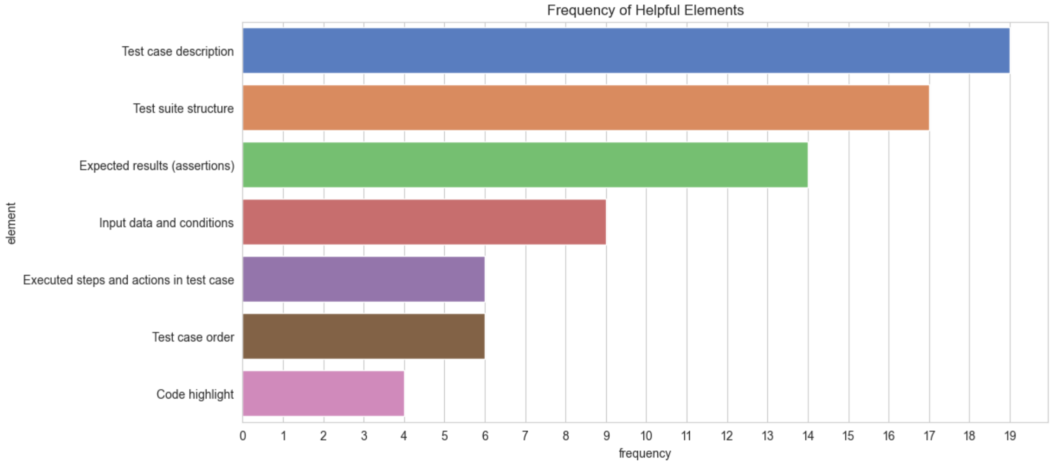


Figure 5.8: Frequency of the helpful elements selected by participants

Figure 5.8 shows the combined results of the question across different treatments. It's clear that participants find the ”Test case description” and the ”Test suite structure” to be the most helpful aspects of the test code. Figure 5.9 shows what participants chose under different treatments. When participants are fixing a test suite with test case clustering, they notice the cluster structure we designed in the test suite and can take advantage of this

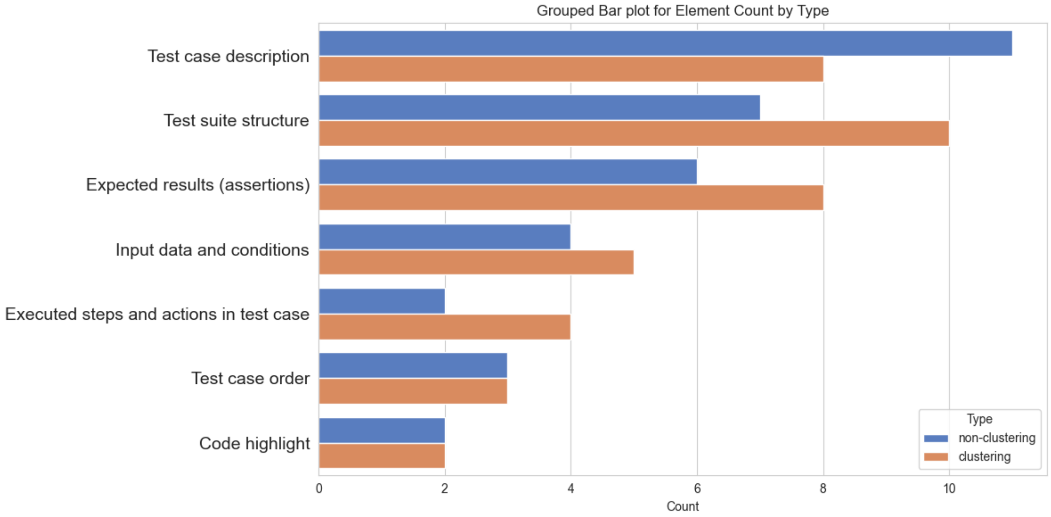


Figure 5.9: Frequency of the helpful elements selected by participants for different treatments

structure. However, when participants are fixing a test suite without test case clustering, they find the ”Test case description” to be the most helpful. This choice makes sense, as many previous studies on source code readability and test code readability have highlighted the importance of test case names in helping participants understand the content of test cases. This also indirectly confirms the validity of our pre-processing steps.

5.2.2 RQ2.2: To what extent does test case clustering impact the developer's ability on identifying potential error scenarios

In the task of identifying potential error scenarios, participants are required to select which input conditions will cause exceptions when calling a specific method. Figure 5.10 shows the comparison results of the effectiveness and efficiency of different treatments under this task.

We observed that participants who read a test suite with test case clustering tend to identify more input conditions that may cause method exceptions than those who do not use test case clustering (mean value of 6.25 vs 5.625). In the clustering group, some participants identified all exception conditions, while no participants in the non-clustering group could identify all options.

In terms of efficiency, the mean value of the clustering group (0.697) is higher than that of the non-clustering group (0.634). However, there is no statistical significance between different test suites, whether in terms of effectiveness or efficiency.

Additionally, as outlined in the task' background introduction (see Appendix A), this study seeks to explore the feasibility of *using automatically generated test suites as 'live documentation.'* We presented participants with three questions related to this idea, as shown in Table 5.1.

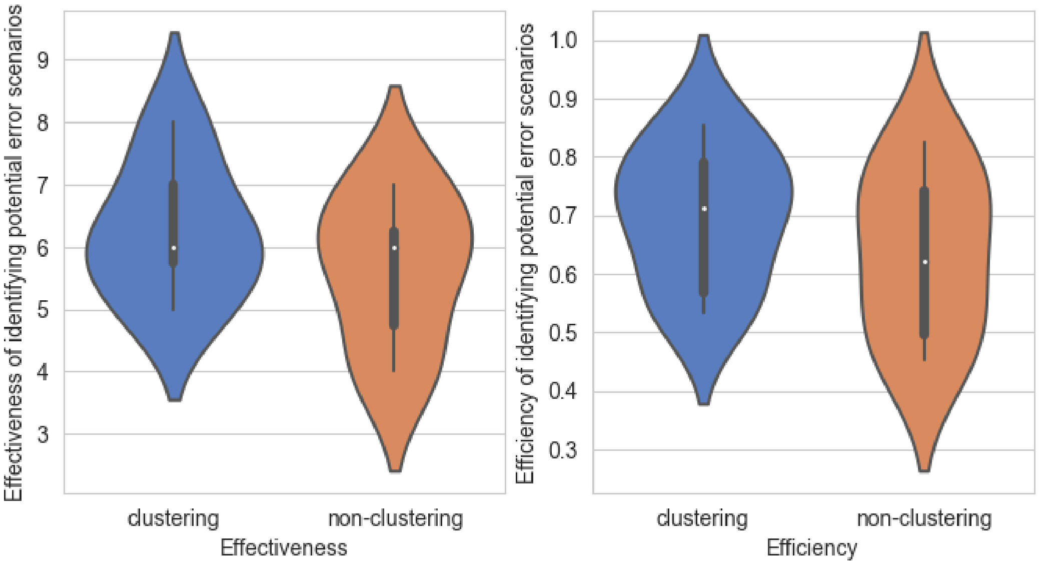


Figure 5.10: Violin plot for effectiveness and efficiency results of task 2

Q1	Do you agree that the test suite provided earlier effectively serves as ”live” documentation that helps you understand these two methods
Q2	Do you agree that it was easy for you to understand the functionality and design of the AnonymousClass from the test suite
Q3	Do you agree that you were confident in your understanding of the AnonymousClass based on the test suite

Table 5.1: Post-task question descriptions

Participant responses are depicted in the Figure 5.11. A significant majority chose ’Agree’ or ’Strongly agree,’ indicating that they can effectively gain information about the class under test from the automatically generated test suites.

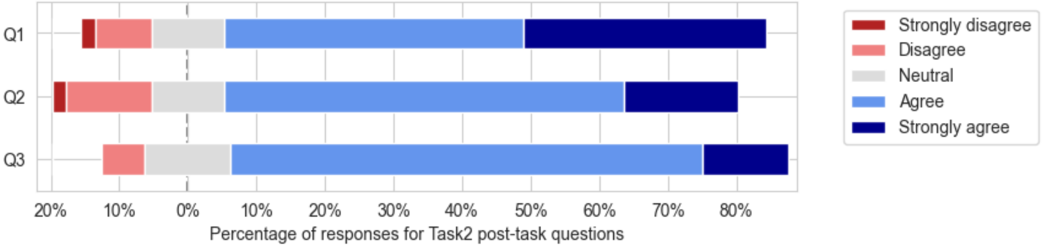


Figure 5.11: 5-point Likert agreement scales results for post-task questions of task 2

Object	Measurement	p-value
`Polygon.js`	effectiveness	0.0789
	efficiency	0.2077
`Queue.js`	effectiveness	0.3701
	efficiency	0.5317
`ShoppingCart.js`	effectiveness	0.3511
	efficiency	0.1003

Table 5.2: P-values from two-way permutation test

5.2.3 Interaction of treatments and subject's experience

The table 4.2 highlights a range in years of JAVASCRIPT programming experience among our participants. This variation could act as a potential factor that might have interaction effect with the independent variable. To determine if subject's experience interact with the treatments on the dependent variables such as effectiveness and efficiency, we conducted an two-way permutation test, focusing on the three mentioned objects. Two-way permutation test is a non-parametric method used to test the null hypothesis that two random variables are exchangeable[18], given the nature of permutation tests, the results can vary across different executions due to the randomness introduced by shuffling. To ensure stability in our results, we set the iterations equal 10000.

The resulting p-values from the tests are presented in the table 5.2. All p-values exceed 0.05. Based on the test results, there is insufficient evidence to assert that there is an interaction effect betweenJAVASCRIPT experience and our treatments.

RQ2: Using test case clustering can significantly improve developer efficiency in related software maintenance tasks when dealing with complex, automatically generated test suites that contain a large number of test cases. However, this improvement is not as noticeable in simpler test suites. Furthermore, using test case clustering in test suites does not improve developer effectiveness in software maintenance tasks. In other words, it does not affect the quality of task completion by developers.

5.3 Threats to Validity

Threats to conclusion validity pertain to factors affecting the ability to make correct statistical inferences regarding the relationship between an experiment's treatment and its outcome[55]. This concept is also known as statistical conclusion validity. To mitigate these threats, we first used the Shapiro-Wilk test[48] to assess whether the data is in normal distribution after data collection. Based on its results, we chose the more versatile non-parametric statistical test, the Wilcoxon rank-sum test[36], with a p-value threshold of 0.05, for comparing experimental outcomes.

Threats to internal validity refer to unnoticed influences on the causal relationship of the independent variable. One such potential threat in our study is the learning effect[55]. As our experiment involved multiple tasks with the same treatments, subjects could experience a learning curve, affecting their reactions to the questions over time. To address this, we used different objects for different tasks and adopted a within-subjects counterbalance design. This ensured subjects did not encounter different treatments for the same object during the experiment. Another threat is subject selection. While we aimed to recruit a set of participants from different background, our final group was not evenly distributed across their backgrounds, potentially making it unrepresentative of the larger population.

Threats to construct validity mainly relate to experimental design[55]. A significant threat in our research is the challenge of adequately defining tasks before translating them into measures or treatments. The core issue is that the comprehensibility of a test suite is not straightforwardly quantifiable. Instead, we gauge it by analyzing subjects' performance on different software tasks, inferring the comprehensibility of one test suite over another. Yet, given our experiment's constraints, we cannot cover all software tasks tied to comprehensibility, potentially affecting our results' validity.

Threats to external validity deal with the generalization of our findings beyond the study's context[55]. Our study only focuses automatically test suite generation tools on JavaScript with a specific testing framework for test case clustering. Hence, our findings might not directly apply to broader software testing studies.

Chapter 6

Discussion

6.1 Findings

Finding 1 *Developers' strongly agree/disagree evaluations of the clustering results stem from the discrepancies in our clustering method's understanding of method invocations.*

In the **RQ1** experiment, we used open-ended questions to analyze why developers saw certain test cases within clusters as similar and why others appeared dissimilar. We also asked about their criteria for manually clustering test cases. The primary takeaway was that most developers consider method invocations in test cases as the most critical factor, with input parameters and output results coming next.

This perspective aligns closely with our test case clustering approach, which emphasizes method invocations (function objectives). In addition, our method takes into account test case branch coverage (branch objectives), a factor related to test case input parameters. Therefore, while no single clustering method for test cases can claim perfection, our objective-based clustering received acceptance from a majority of the participants during experiment. In the case of ”cluster2” in figure 5.4, when two simple test cases had identical method invocations, developers strongly agreed with our clustering. However, for ”cluster6” in figure 5.3, our clustering method identified the test cases as same cluster because invoking `addItem` would also trigger `validateInput`. But developers often differentiate based on the primary method invocation, which explains the many ’disagree’ evaluations for ”cluster6” when the source code is not well-understood.

A recurring theme that led developers to question our clustering results concerns the sequence of method calls. As an example, with two methods—`addItem` and `removeItem`—if the sequence of their invocation does not alter the branch objectives, our clustering method would see test cases invoking `addItem` followed by `removeItem`, and vice versa, as similar. Yet, developers often view the order of method calls as crucial, indicating the test case's true intent, they will definitely disagree with the clustering results for this example.

The divergence in understanding method invocations is not just between our clustering method and developers; it also exists between automatically generated test cases and manually written test cases. This difference partly explains why some developers remain reticent about adopting automated test case generation tools.

Finding 2 *The complexity of a test suite magnifies the impact of test case clustering on its comprehensibility.*

From the results of Task 1, it is evident that for the `Polygon.js` object, there was a significant difference in the efficiency (time) developers spent on fault localization tasks when using a test suite that employed clustering compared to one that did not. Conversely, for the `Queue.js` object, the differences between the treatments were not statistically significant.

The observed variation may be attributed to the complexity of the software objects in task. For instance, relatively simple objects, such as `Queue.js`, with a limited number of test cases might not fully benefit from clustering operation. A smaller number of test cases can lead to less cognitive overhead, allowing developers to grasp the entirety of the test suite context without relying heavily on external tools or processes. Consequently, the potential advantages of clustering in aiding fault localization become less conspicuous.

However, as the complexity of an object increases, relying on memory alone to understand the full context of the test suite becomes less feasible. In such scenarios, the organization and structuring of the test cases are crucial for improving their comprehensibility, subsequently influencing the efficacy of fault localization task.

Finding 3 *Test case description is the most intuitive and cost-effective method to enhance comprehensibility.*

While our research predominantly focuses on the impact of test case clustering on test suite comprehensibility, the fundamental role of descriptive, natural language-like test case descriptions cannot be overlooked. Such descriptions aid developers in quickly grasping the purpose and scope of each test case. Furthermore, they offer a cost-effective approach, eliminating the need for extensive documentation or additional tools.

If we had chosen not to refine the original test case descriptions during preprocessing, the participants might have spent extra efforts to understand the test suite and encountered significant challenges during the experiments, because reading is the cornerstone of code comprehension. This perspective is supported by the outcomes of Task 1, as illustrated in Figures 5.8 and 5.9, where participants identified the test case description as the most beneficial element, followed closely by the test suite structure, or test case clustering.

In recent program comprehension research, many scholars are harnessing the power Large Language Model (LLM) to produce more accurate test case summarization and identifier names for test cases. Implementing such strategies bolsters developers' ability to understand the code. Thus, the utilization of LLM to augment the readability and comprehensibility of automatically generated test suites emerges as a compelling direction for upcoming research.

Finding 4 *Automatically generated tests as living documentation.*

Modern programming paradigms, such as Agile and Extreme Programming (XP), often emphasize working code over extensive documentation. Instead, they advocate tests as a form of ”living documentation.”

Traditionally, this idea was associated primarily with manually created test suites. In our research, we sought to explore the feasibility of utilizing automatically generated test code as ”living documentation.”. In Task 2, participants were presented solely with automatically generated test code. Their task was to interpret the correct input-output relationships and potential functionalities of a specific method after reviewing the test code. Once exposed to the source code of the class under tested, participants compared their initial understanding of the method, based on the test code, to its true functionality. Based on the results in Figure 5.11, the majority of participants agreed that the provided test suite effectively functioned as ”live” documentation, enhancing their grasp of the methods in question.

We believe this finding stem from two primary factors: the highly structural coverage offered by automatic test case generation tools and the improved readability from our preprocessing. Highly structural coverage ensures the generated test suites capture the primary use-cases for a specific method, and preprocessing aids participants in understanding the specific nuances of the class under test.

6.2 Limitations

Throughout our study’s journey, we identified several limitations:

- **Objects**: While our test case clustering method shows promise primarily with JavaScript class modules, it faces challenges in the broader JavaScript ecosystem. Many packages are designed as function modules, where our clustering approach tends to isolate each test case into separate clusters. This outcome runs counter to the very goal of clustering.

- **Preprocessing**: Our current preprocessing operations to improve test case readability are kind of simple, leveraging basic static and dynamic analysis methods. In contrast, leading-edge research today utilize Large Language Models (LLM) to elevate readability.

- **Generalization**: While understanding the theoretical underpinning of test case clustering is simple, its practical application is more intricate. Popular programming languages, including Java and Python, do not offer substantial support in their testing frameworks for this clustering-like approach. As a result, the real-world application of test case clustering across various languages remains a complex endeavor, despite its potential benefits for comprehensibility.

6.3 Future works

Considering our findings and the outlined limitations, several opportunities arise for advancing our research:

Integration with SynTest-Framework: Currently, our approach relies solely on the automatically generated test cases from SynTest-JavaScript, with both preprocessing and clustering functioning independently of it. However, given the adaptability of the SynTest-Framework, facets of our project could be integrated seamlessly.

Harnessing LLM: Deploying a fine-tuned Large Language Model (LLM) holds potential. Such a model can produce enhanced test case descriptions, refine variable naming within test cases, and even generate comprehensive test case documentation, elevating readability.

Extending to other programming languages: Both Java and Python have their own test case generation tools such as EvoSuite and Pynguin, respectively. Since these tools create unit tests using similar algorithms, DynaMOSA, our clustering approach should be theoretically transferrable to automatically generated test cases across different programming languages.

Broader empirical evaluation: Evaluating a broader and more diverse set of objects would be beneficial. The assessment in this study exclusively focused on three different objects. Despite our efforts to select a diverse range of categories, a wider evaluation could yield more definitive conclusions. Additionally, experiments focusing on more complex objects would further validate the insights from our finding **Finding 2**.

Chapter 7

Conclusions

In this thesis, we evaluated test case similarity based on covered objectives, using these as the basis for a clustering method designed specifically for automatically generated test cases. These objectives aligned with the function and branch objectives used by automated test case generation tools during their search process of generating the test cases.

We undertook empirical research to examine the viability of test case clustering and its impact on test suite comprehensibility. Given the abstract nature of comprehensibility, we deployed various software maintenance tasks to indirectly measure it by monitoring developer effectiveness and efficiency.

Our primary results addressed two key queries: (i) The extent of developer agreement with our clustering method. (ii) The impact of test case clustering on the test suite. The results revealed that: (i) Most developers found our clustering method apt for grouping similar test cases. (ii) For complex test suites, test case clustering enhanced efficiency by reducing the time developers spent on software maintenance tasks. In essence, under particular conditions, test case clustering positively influences test suite comprehensibility. Additionally, we explored broader topics, emphasizing the pivotal role of test case descriptions in test suite comprehension and the potential of automatically generated test suites to act as ”living documentation.”

As outlined in Chapter 1, this thesis delves into the comprehensibility of automatically generated test code at the test suite level. We have found that test case clustering significantly aids developers in comprehending test suites. The relevance of our results spans beyond just JavaScript test suites, extending to any test suite automatically generated from Search-Based Software Testing (SBST) in other programming languages. Ultimately, our findings pave the way for developers to more effectively harness automated test case generation tools.

Bibliography

[1] Enterprise online survey software tools - alchemer. `https://www.alchemer.com/`. [Accessed 24-08-2023].

[2] Mocha - the fun, simple, flexible JavaScript test framework — mochajs.org. `https://mochajs.org`. [Accessed 24-08-2023].

[3] RSpec: Behaviour Driven Development for Ruby — rspec.info. `https://rspec.info/`. [Accessed 24-08-2023].

[4] Maha Alkhayrat, Mohamad Aljnidi, and Kadan Aljoumaa. A comparative dimensionality reduction study in telecom customer segmentation using deep learning and pca. *Journal of Big Data*, 7:1–23, 2020.

[5] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of systems and software*, 86(8):1978–2001, 2013.

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[7] Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[8] Benjamin S Bloom, Max D Engelhart, Edward J Furst, Walker H Hill, and David R Krathwohl. *Taxonomy of educational objectives: The classification of educational goals. Handbook 1: Cognitive domain*. McKay New York, 1956.

[9] Frederick P Brooks Jr. *The mythical man-month: essays on software engineering*. Pearson Education, 1995.

[10] Raymond PL Buse and Westley R Weimer. Learning a metric for code readability. *IEEE Transactions on software engineering*, 36(4):546–558, 2009.

[11] Mariano Ceccato, Alessandro Marchetto, Leonardo Mariani, Cu D Nguyen, and Paolo Tonella. Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):1–38, 2015.

[12] Tsong Yueh Chen, Hing Leung, and Ieng Kei Mak. Adaptive random testing. In *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making: 9th Asian Computing Science Conference. Dedicated to Jean-Louis Lassez on the Occasion of His 5th Birthday. Chiang Mai, Thailand, December 8-10, 2004. Proceedings 9*, pages 320–329. Springer, 2005.

[13] Nour Chetouane, Franz Wotawa, Hermann Felbinger, and Mihai Nica. On using k-means clustering for test suite reduction. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 380–385. IEEE, 2020.

[14] Bas Cornelissen, Arie Van Deursen, Leon Moonen, and Andy Zaidman. Visualizing testsuites to aid in software understanding. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 213–222. IEEE, 2007.

[15] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 107–118, 2015.

[16] Ermira Daka, José Miguel Rojas, and Gordon Fraser. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 57–67, 2017.

[17] Richard O Duda, Peter E Hart, et al. *Pattern classification and scene analysis*, volume 3. Wiley New York, 1973.

[18] Eugene Edgington and Patrick Onghena. *Randomization tests*. CRC press, 2007.

[19] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.

[20] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *Proceedings of the 26th Conference on Program Comprehension*, pages 286–296, 2018.

[21] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[22] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2012.

[23] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical software engineering*, 20:611–639, 2015.

[24] Wenhao Fu, Huiqun Yu, Guisheng Fan, and Xiang Ji. Coverage-based clustering and scheduling approach for test case prioritization. *IEICE TRANSACTIONS on Information and Systems*, 100(6):1218–1230, 2017.

[25] Ursula Fuller, Colin G Johnson, Tuukka Ahoniemi, Diana Cukierman, Isidoro Hernán-Losada, Jana Jackova, Essi Lahtinen, Tracy L Lewis, Donna McGee Thompson, Charles Riedesel, et al. Developing a computer science-specific learning taxonomy. *ACm SIGCSE Bulletin*, 39(4):152–170, 2007.

[26] Michaela Greiler, Arie van Deursen, and Andy Zaidman. Measuring test case similarity to support test suite understanding. In *Objects, Models, Components, Patterns: 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings 50*, pages 91–107. Springer, 2012.

[27] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12. IEEE, 2015.

[28] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[29] Johannes Hofmeister, Janet Siegmund, and Daniel V Holt. Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER)*, pages 217–227. IEEE, 2017.

[30] Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. Effective and efficient api misuse detection via exception propagation and search-based testing. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 192–203, 2019.

[31] Manju Khari and Prabhat Kumar. An extensive evaluation of search-based software testing: a review. *Soft Computing*, 23:1933–1946, 2019.

[32] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931, 2023. doi: 10.1109/ICSE48619.2023.00085.

[33] Linyi Li, Zhenwen Li, Weijie Zhang, Jun Zhou, Pengcheng Wang, Jing Wu, Guanghua He, Xia Zeng, Yuetang Deng, and Tao Xie. Clustering test steps in natural language toward automating test automation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1285–1295, 2020.

[34] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172, 2022.

[35] T Soni Madhulatha. An overview on clustering methods. *arXiv preprint arXiv:1205.1117*, 2012.

[36] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[37] Phil McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[38] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. Evaluating code readability and legibility: An examination of human-centric studies. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 348–359. IEEE, 2020.

[39] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

[40] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*, pages 1–10. IEEE, 2015.

[41] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2017.

[42] Lior Rokach and Oded Maimon. Clustering methods. *Data mining and knowledge discovery handbook*, pages 321–352, 2005.

[43] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[44] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. Deeptc-enhancer: Improving the readability of automatically generated tests. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 287–298, 2020.

[45] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability: How far are we? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 417–427. IEEE, 2017.

[46] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. Comprehending studies on program comprehension. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 308–311. IEEE, 2017.

[47] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. Adaptive test generation using a large language model, 2023.

[48] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[49] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. Exploring the effectiveness of large language models in generating unit tests, 2023.

[50] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. Guess what: Test case generation for javascript with unsupervised probabilistic type inference. In *Search-Based Software Engineering: 14th International Symposium, SSBSE 2022, Singapore, November 17–18, 2022, Proceedings*, pages 67–82. Springer, 2022.

[51] Robert L Thorndike. Who belongs in the family? *Psychometrika*, 18(4):267–276, 1953.

[52] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context, 2021.

[53] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[54] Markos Viggiato, Dale Paas, Chris Buzon, and Cor-Paul Bezemer. Identifying similar test cases that are specified in natural language. *IEEE Transactions on Software Engineering*, 49(3):1027–1043, 2022.

[55] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[56] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[57] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 201–212, 2009.

[58] Marcel Zalmanovici, Orna Raz, and Rachel Tzoref-Brill. Cluster-based test suite functional analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 962–967, 2016.

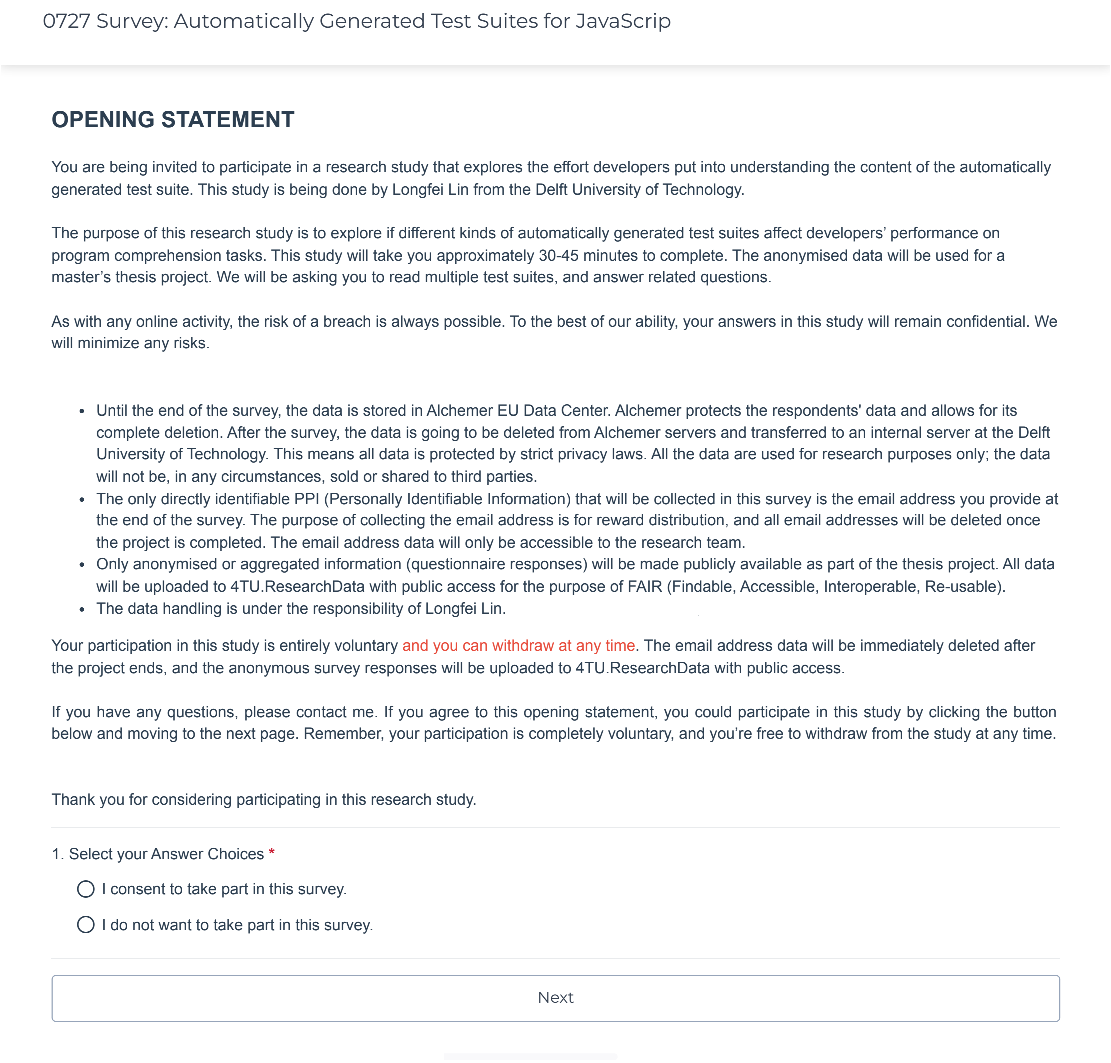
[59] Sai Zhang. Practical semantic test simplification. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1173–1176. IEEE, 2013.

Appendix A

Online Survey Demo

Presented herein is the comprehensive content of the online survey used in our experiment.

A. Online Survey Demo



0727 Survey: Automatically Generated Test Suites for JavaScrip

OPENING STATEMENT

You are being invited to participate in a research study that explores the effort developers put into understanding the content of the automatically generated test suite. This study is being done by Longfei Lin from the Delft University of Technology.

The purpose of this research study is to explore if different kinds of automatically generated test suites affect developers' performance on program comprehension tasks. This study will take you approximately 30-45 minutes to complete. The anonymised data will be used for a master's thesis project. We will be asking you to read multiple test suites, and answer related questions.

As with any online activity, the risk of a breach is always possible. To the best of our ability, your answers in this study will remain confidential. We will minimize any risks.

- Until the end of the survey, the data is stored in Alchemer EU Data Center. Alchemer protects the respondents' data and allows for its complete deletion. After the survey, the data is going to be deleted from Alchemer servers and transferred to an internal server at the Delft University of Technology. This means all data is protected by strict privacy laws. All the data are used for research purposes only; the data will not be, in any circumstances, sold or shared to third parties.
- The only directly identifiable PPI (Personally Identifiable Information) that will be collected in this survey is the email address you provide at the end of the survey. The purpose of collecting the email address is for reward distribution, and all email addresses will be deleted once the project is completed. The email address data will only be accessible to the research team.
- Only anonymised or aggregated information (questionnaire responses) will be made publicly available as part of the thesis project. All data will be uploaded to 4TU.ResearchData with public access for the purpose of FAIR (Findable, Accessible, Interoperable, Re-usable).
- The data handling is under the responsibility of Longfei Lin.

Your participation in this study is entirely voluntary and you can withdraw at any time. The email address data will be immediately deleted after the project ends, and the anonymous survey responses will be uploaded to 4TU.ResearchData with public access.

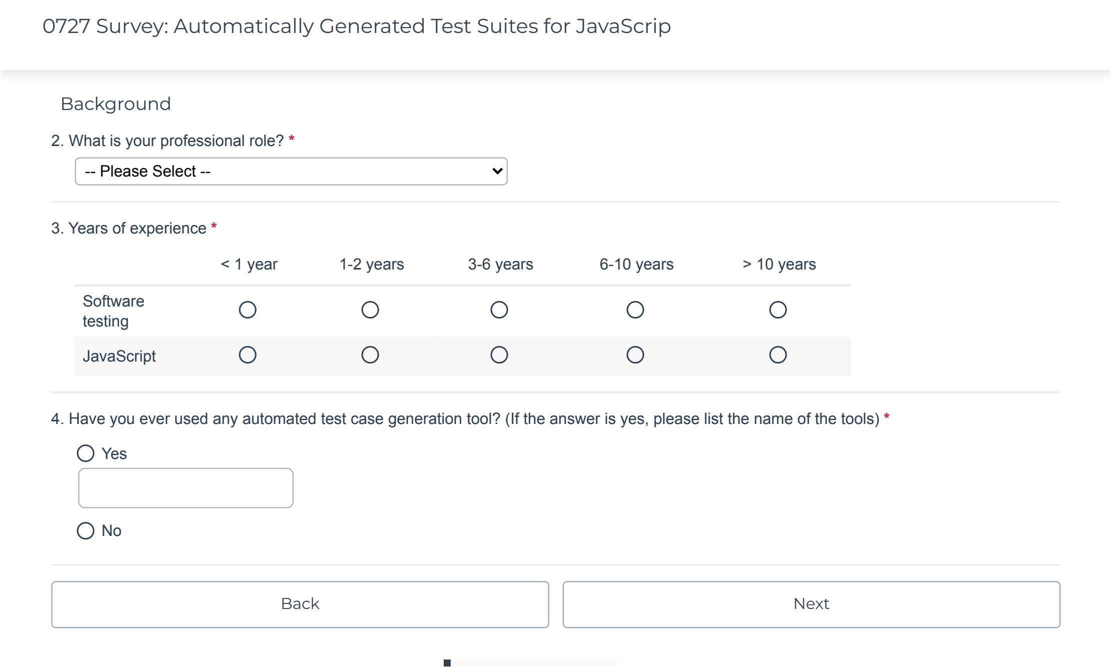
If you have any questions, please contact me. If you agree to this opening statement, you could participate in this study by clicking the button below and moving to the next page. Remember, your participation is completely voluntary, and you're free to withdraw from the study at any time.

Thank you for considering participating in this research study.

1. Select your Answer Choices *

○ I consent to take part in this survey.

○ I do not want to take part in this survey.



0727 Survey: Automatically Generated Test Suites for JavaScrip

Background

2. What is your professional role? *

-- Please Select --

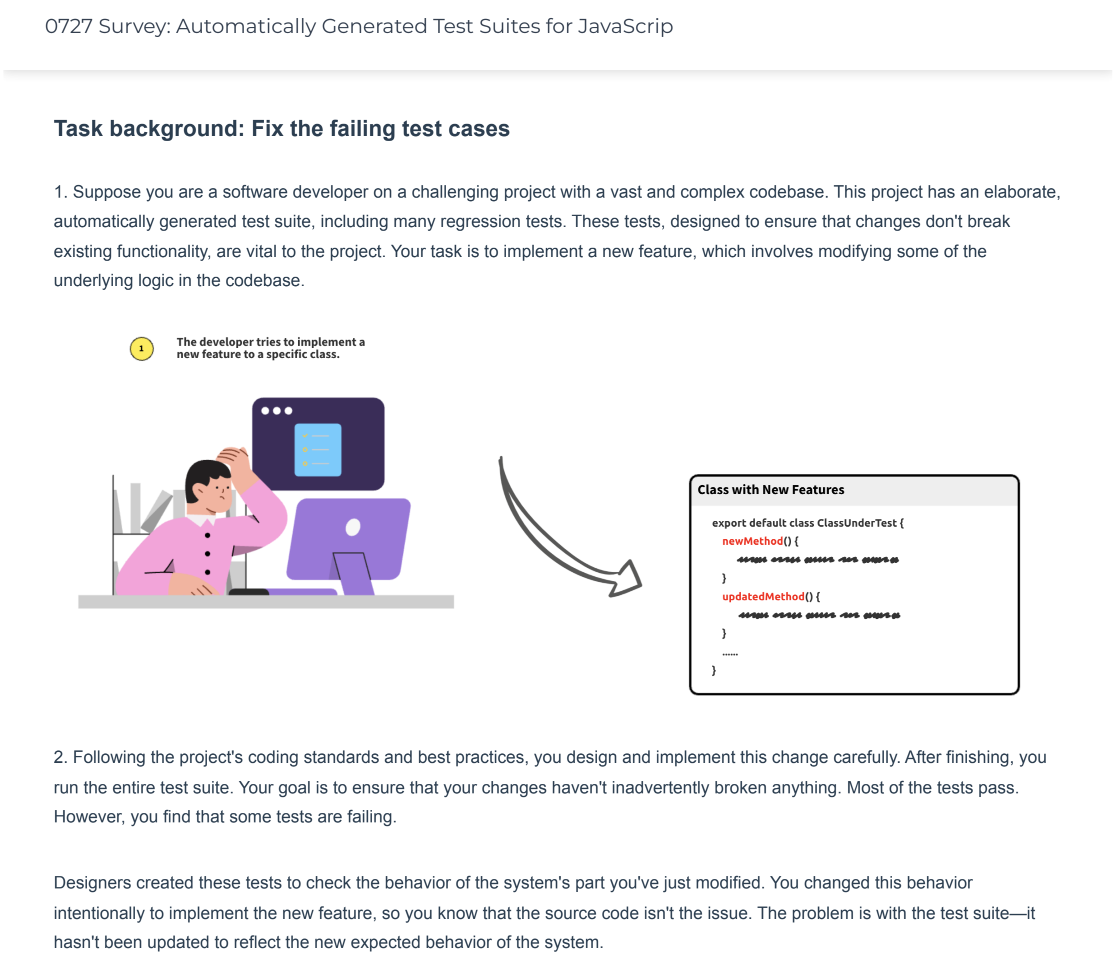
3. Years of experience *

	< 1 year	1-2 years	3-6 years	6-10 years	> 10 years
Software testing	○	○	○	○	○
JavaScript	○	○	○	○	○

4. Have you ever used any automated test case generation tool? (If the answer is yes, please list the name of the tools) *

○ Yes

○ No



0727 Survey: Automatically Generated Test Suites for JavaScrip

Task background: Fix the failing test cases

1. Suppose you are a software developer on a challenging project with a vast and complex codebase. This project has an elaborate, automatically generated test suite, including many regression tests. These tests, designed to ensure that changes don't break existing functionality, are vital to the project. Your task is to implement a new feature, which involves modifying some of the underlying logic in the codebase.

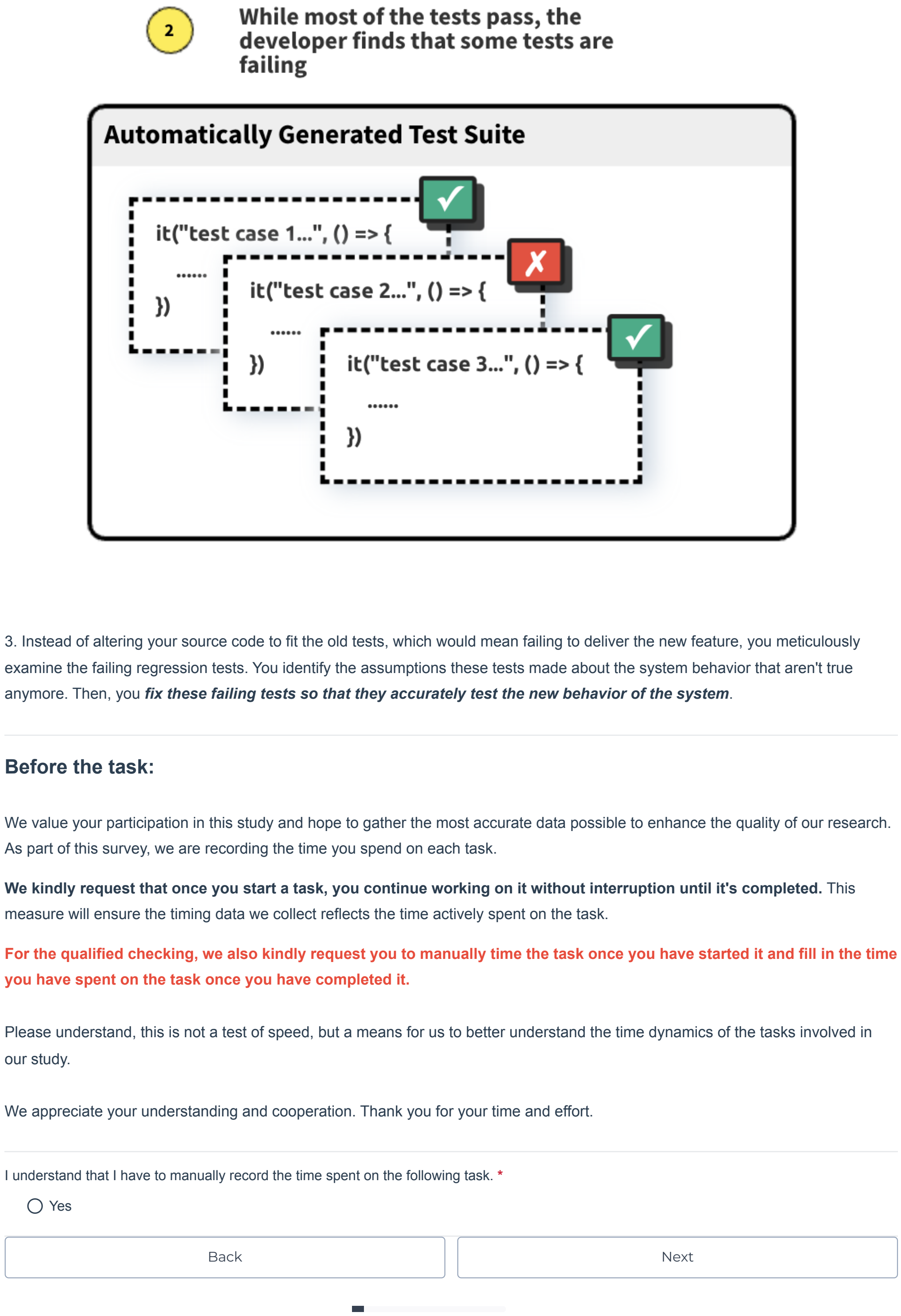
1 The developer tries to implement a new feature to a specific class.

Class with New Features

```
export default class ClassUnderTest {
  newMethod() {
     [illegible]
  }
  updatedMethod() {
     [illegible]
  }
  ......
}
```

2. Following the project's coding standards and best practices, you design and implement this change carefully. After finishing, you run the entire test suite. Your goal is to ensure that your changes haven't inadvertently broken anything. Most of the tests pass. However, you find that some tests are failing.

Designers created these tests to check the behavior of the system's part you've just modified. You changed this behavior intentionally to implement the new feature, so you know that the source code isn't the issue. The problem is with the test suite—it hasn't been updated to reflect the new expected behavior of the system.



2 While most of the tests pass, the developer finds that some tests are failing

Automatically Generated Test Suite

```
it("test case 1...", () => {
  ......
})
```

✓

```
it("test case 2...", () => {
  ......
})
```

✗

```
it("test case 3...", () => {
  ......
})
```

✓

3. Instead of altering your source code to fit the old tests, which would mean failing to deliver the new feature, you meticulously examine the failing regression tests. You identify the assumptions these tests made about the system behavior that aren't true anymore. Then, you ***fix these failing tests so that they accurately test the new behavior of the system***.

Before the task:

We value your participation in this study and hope to gather the most accurate data possible to enhance the quality of our research. As part of this survey, we are recording the time you spend on each task.

We kindly request that once you start a task, you continue working on it without interruption until it's completed. This measure will ensure the timing data we collect reflects the time actively spent on the task.

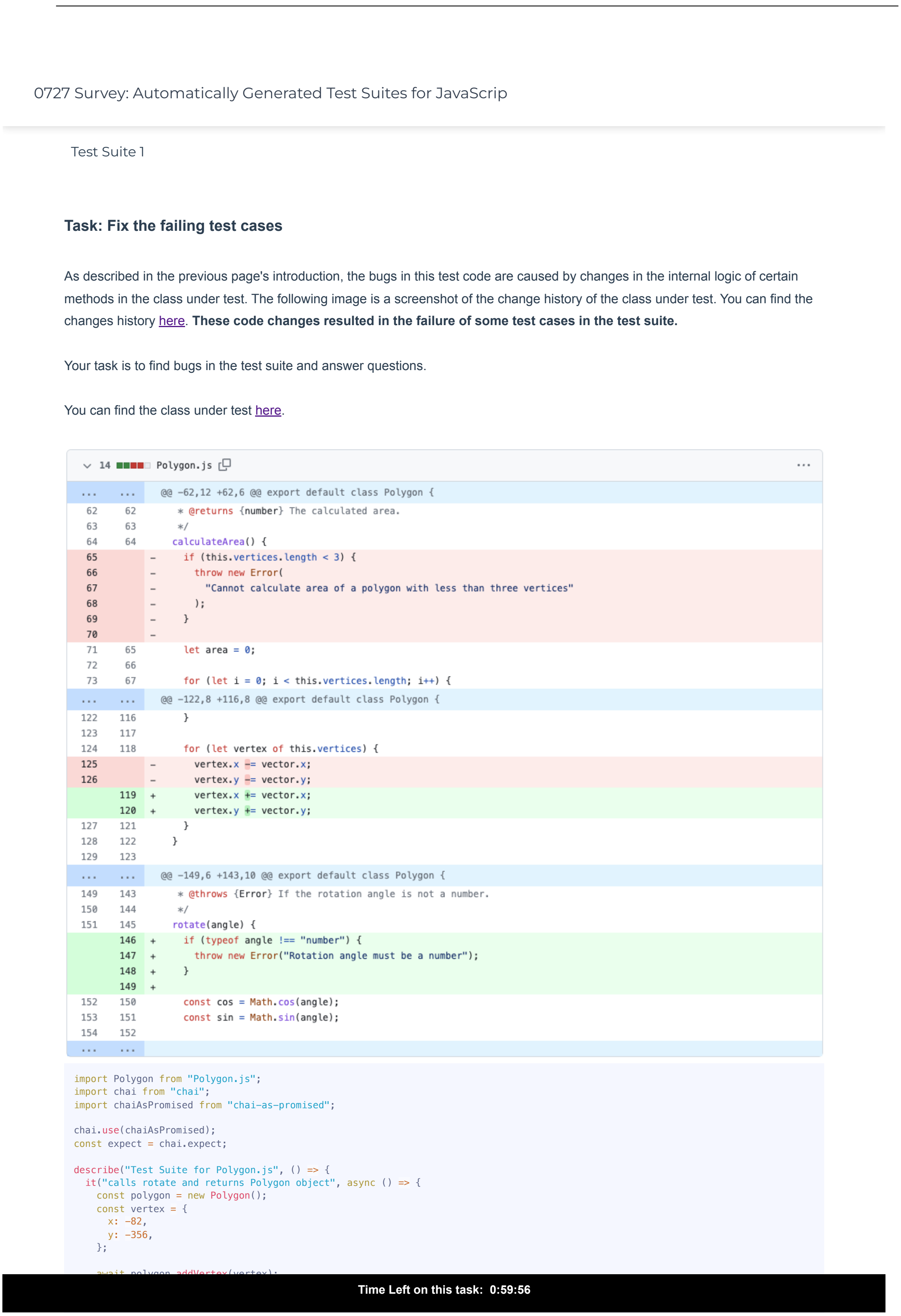
For the qualified checking, we also kindly request you to manually time the task once you have started it and fill in the time you have spent on the task once you have completed it.

Please understand, this is not a test of speed, but a means for us to better understand the time dynamics of the tasks involved in our study.

We appreciate your understanding and cooperation. Thank you for your time and effort.

I understand that I have to manually record the time spent on the following task. *

○ Yes



0727 Survey: Automatically Generated Test Suites for JavaScrip

Test Suite 1

Task: Fix the failing test cases

As described in the previous page's introduction, the bugs in this test code are caused by changes in the internal logic of certain methods in the class under test. The following image is a screenshot of the change history of the class under test. You can find the changes history here. **These code changes resulted in the failure of some test cases in the test suite.**

Your task is to find bugs in the test suite and answer questions.

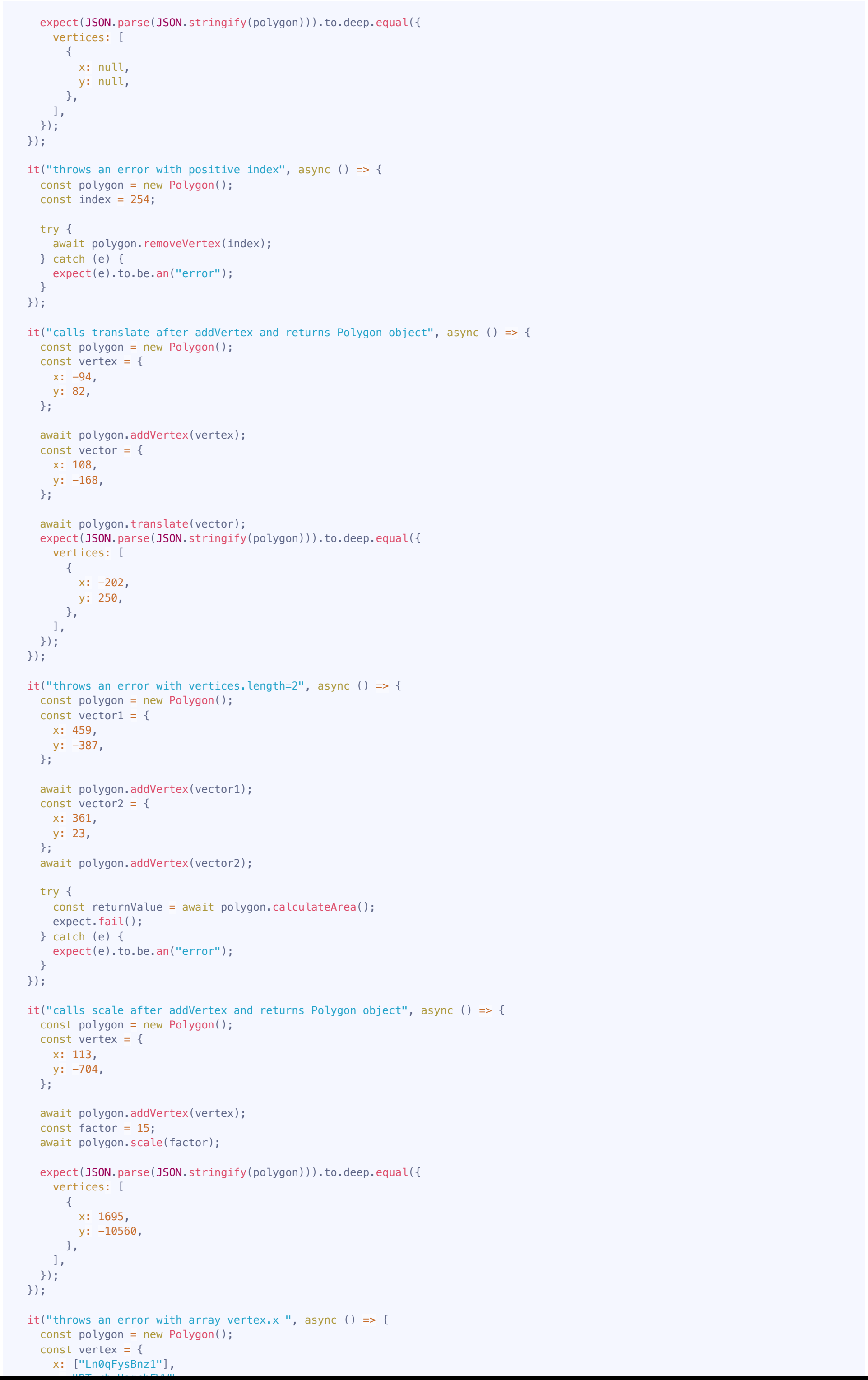
You can find the class under test here.

14 Polygon.js

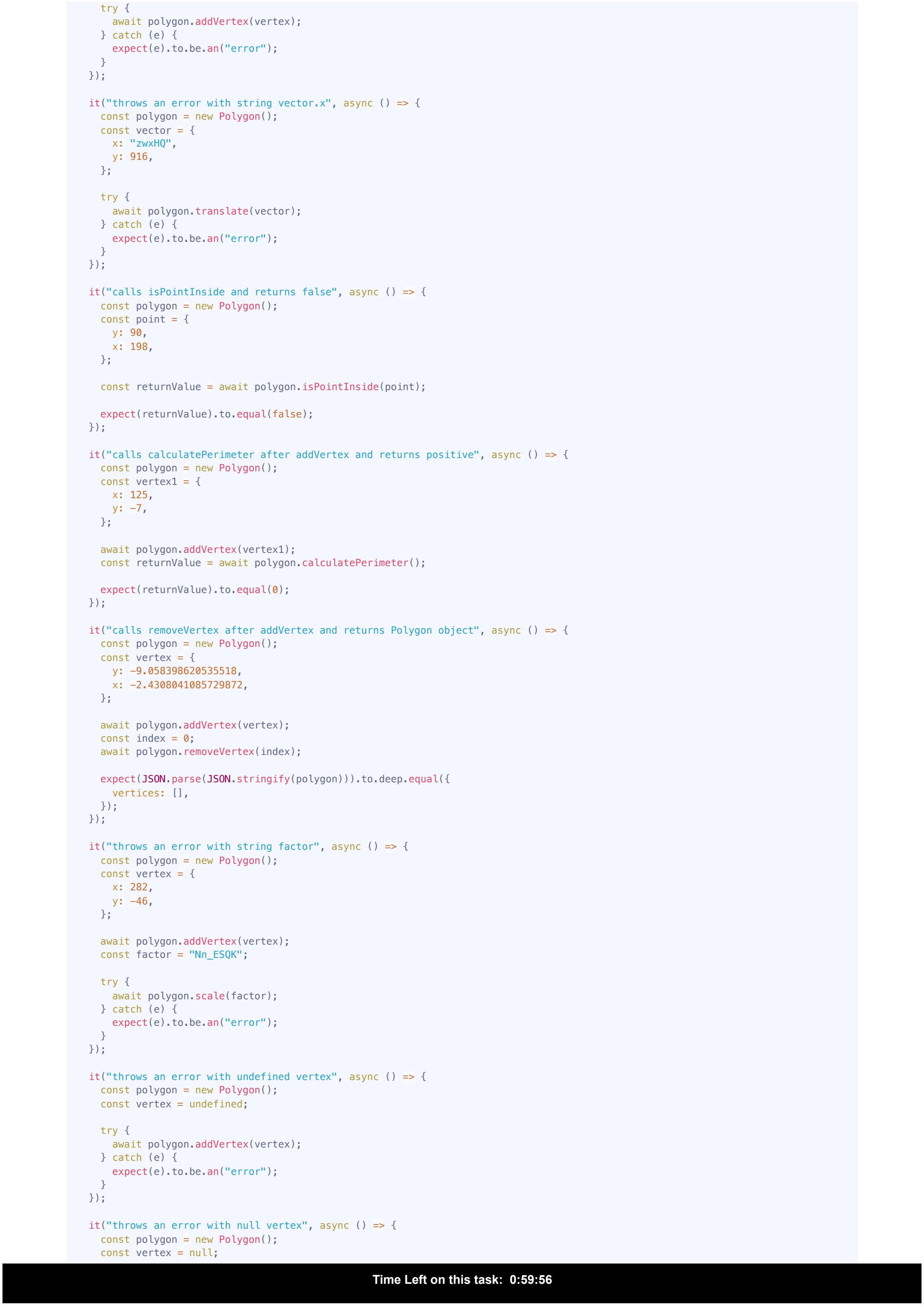
```diff
@@ -62,12 +62,6 @@ export default class Polygon {
     * @returns {number} The calculated area.
     */
    calculateArea() {
-     if (this.vertices.length < 3) {
-       throw new Error(
-         "Cannot calculate area of a polygon with less than three vertices"
-       );
-     }
-
      let area = 0;

      for (let i = 0; i < this.vertices.length; i++) {
@@ -122,8 +116,8 @@ export default class Polygon {
      }

      for (let vertex of this.vertices) {
-       vertex.x -= vector.x;
-       vertex.y -= vector.y;
+       vertex.x += vector.x;
+       vertex.y += vector.y;
      }
    }

@@ -149,6 +143,10 @@ export default class Polygon {
     * @throws {Error} If the rotation angle is not a number.
     */
    rotate(angle) {
+     if (typeof angle !== "number") {
+       throw new Error("Rotation angle must be a number");
+     }
+
      const cos = Math.cos(angle);
      const sin = Math.sin(angle);

```

```javascript
import Polygon from "Polygon.js";
import chai from "chai";
import chaiAsPromised from "chai-as-promised";

chai.use(chaiAsPromised);
const expect = chai.expect;

describe("Test Suite for Polygon.js", () => {
  it("calls rotate and returns Polygon object", async () => {
    const polygon = new Polygon();
    const vertex = {
      x: -82,
      y: -356,
    };

    await polygon.addVertex(vertex);
```

Time Left on this task: 0:59:56

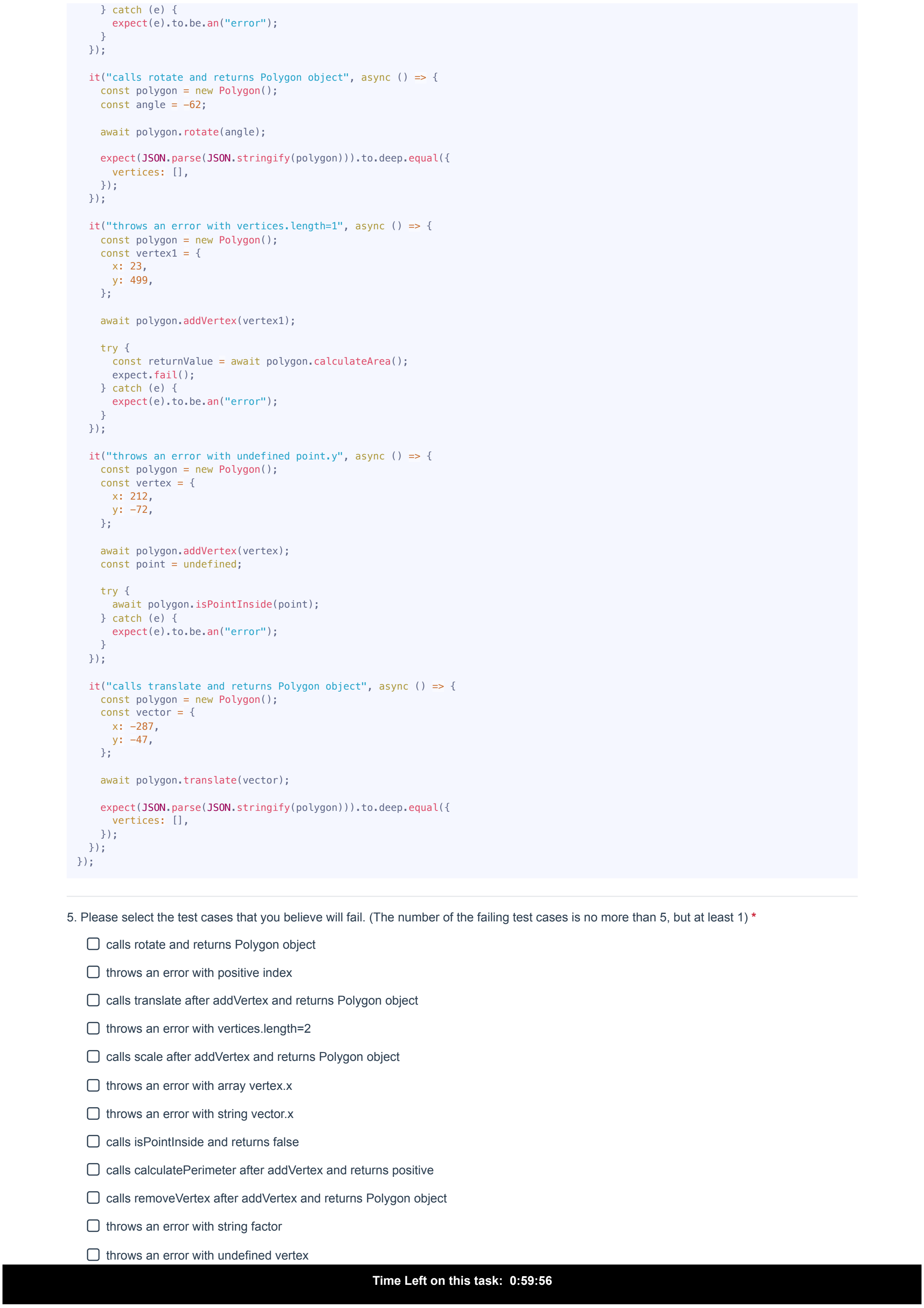


```
    expect(JSON.parse(JSON.stringify(polygon))).to.deep.equal({
      vertices: [
        {
          x: null,
          y: null,
        },
      ],
    });
  });

  it("throws an error with positive index", async () => {
    const polygon = new Polygon();
    const index = 254;

    try {
      await polygon.removeVertex(index);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("calls translate after addVertex and returns Polygon object", async () => {
    const polygon = new Polygon();
    const vertex = {
      x: -94,
      y: 82,
    };

    await polygon.addVertex(vertex);
    const vector = {
      x: 108,
      y: -168,
    };

    await polygon.translate(vector);
    expect(JSON.parse(JSON.stringify(polygon))).to.deep.equal({
      vertices: [
        {
          x: -202,
          y: 250,
        },
      ],
    });
  });

  it("throws an error with vertices.length=2", async () => {
    const polygon = new Polygon();
    const vector1 = {
      x: 459,
      y: -387,
    };

    await polygon.addVertex(vector1);
    const vector2 = {
      x: 361,
      y: 23,
    };
    await polygon.addVertex(vector2);

    try {
      const returnValue = await polygon.calculateArea();
      expect.fail();
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("calls scale after addVertex and returns Polygon object", async () => {
    const polygon = new Polygon();
    const vertex = {
      x: 113,
      y: -704,
    };

    await polygon.addVertex(vertex);
    const factor = 15;
    await polygon.scale(factor);

    expect(JSON.parse(JSON.stringify(polygon))).to.deep.equal({
      vertices: [
        {
          x: 1695,
          y: -10560,
        },
      ],
    });
  });

  it("throws an error with array vertex.x ", async () => {
    const polygon = new Polygon();
    const vertex = {
      x: ["Ln0qFysBnz1"],
```



```
    try {
      await polygon.addVertex(vertex);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error with string vector.x", async () => {
    const polygon = new Polygon();
    const vector = {
      x: "zwxHQ",
      y: 916,
    };

    try {
      await polygon.translate(vector);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("calls isPointInside and returns false", async () => {
    const polygon = new Polygon();
    const point = {
      y: 90,
      x: 198,
    };

    const returnValue = await polygon.isPointInside(point);

    expect(returnValue).to.equal(false);
  });

  it("calls calculatePerimeter after addVertex and returns positive", async () => {
    const polygon = new Polygon();
    const vertex1 = {
      x: 125,
      y: -7,
    };

    await polygon.addVertex(vertex1);
    const returnValue = await polygon.calculatePerimeter();

    expect(returnValue).to.equal(0);
  });

  it("calls removeVertex after addVertex and returns Polygon object", async () => {
    const polygon = new Polygon();
    const vertex = {
      y: -9.058398620535518,
      x: -2.4308041085729872,
    };

    await polygon.addVertex(vertex);
    const index = 0;
    await polygon.removeVertex(index);

    expect(JSON.parse(JSON.stringify(polygon))).to.deep.equal({
      vertices: [],
    });
  });

  it("throws an error with string factor", async () => {
    const polygon = new Polygon();
    const vertex = {
      x: 282,
      y: -46,
    };

    await polygon.addVertex(vertex);
    const factor = "Nn_ESQK";

    try {
      await polygon.scale(factor);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error with undefined vertex", async () => {
    const polygon = new Polygon();
    const vertex = undefined;

    try {
      await polygon.addVertex(vertex);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error with null vertex", async () => {
    const polygon = new Polygon();
    const vertex = null;
```

Time Left on this task: 0:59:56

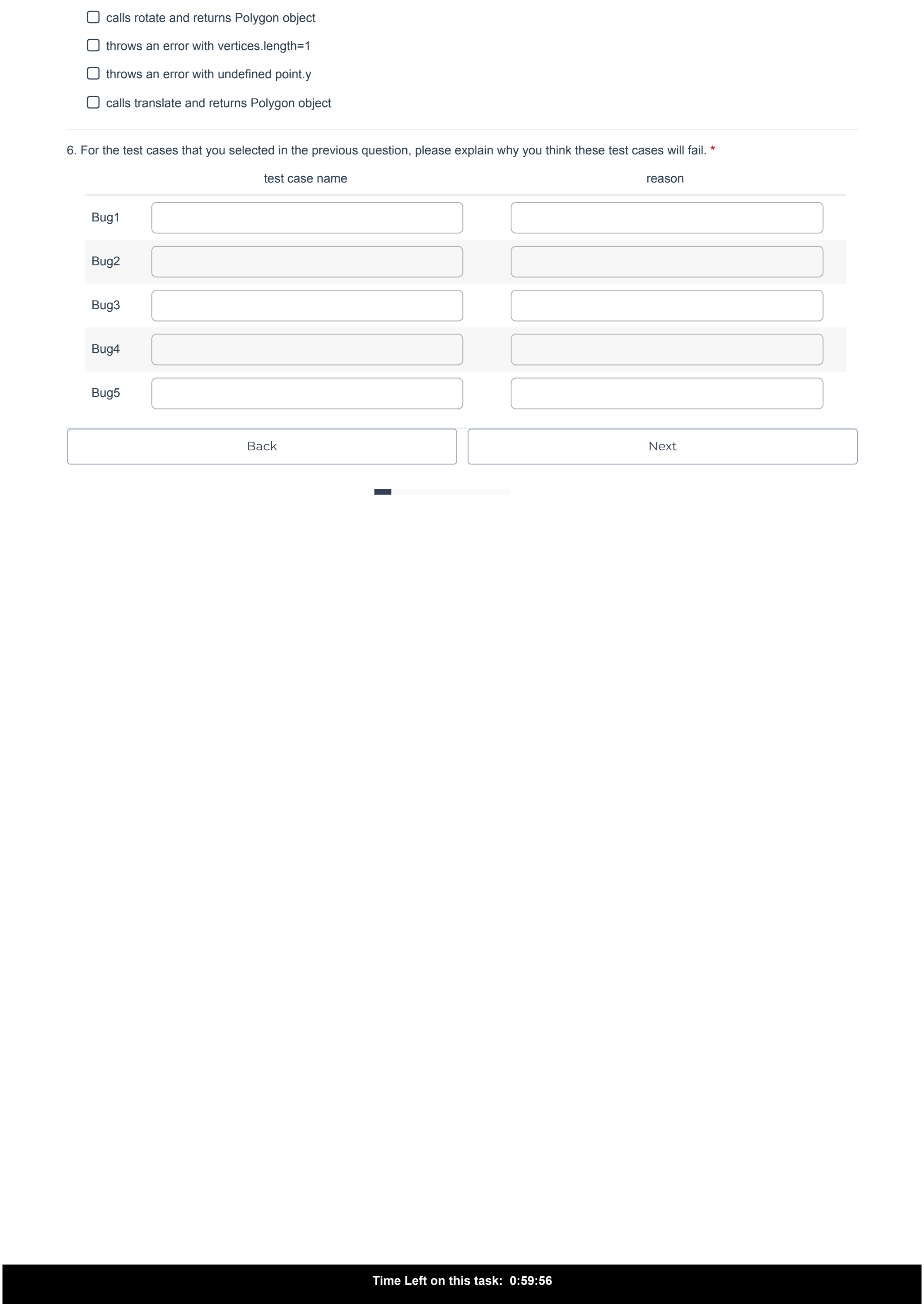


```
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("calls rotate and returns Polygon object", async () => {
    const polygon = new Polygon();
    const angle = -62;

    await polygon.rotate(angle);

    expect(JSON.parse(JSON.stringify(polygon))).to.deep.equal({
      vertices: [],
    });
  });

  it("throws an error with vertices.length=1", async () => {
    const polygon = new Polygon();
    const vertex1 = {
      x: 23,
      y: 499,
    };

    await polygon.addVertex(vertex1);

    try {
      const returnValue = await polygon.calculateArea();
      expect.fail();
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error with undefined point.y", async () => {
    const polygon = new Polygon();
    const vertex = {
      x: 212,
      y: -72,
    };

    await polygon.addVertex(vertex);
    const point = undefined;

    try {
      await polygon.isPointInside(point);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("calls translate and returns Polygon object", async () => {
    const polygon = new Polygon();
    const vector = {
      x: -287,
      y: -47,
    };

    await polygon.translate(vector);

    expect(JSON.parse(JSON.stringify(polygon))).to.deep.equal({
      vertices: [],
    });
  });
});
```

5. Please select the test cases that you believe will fail. (The number of the failing test cases is no more than 5, but at least 1) *

- □ calls rotate and returns Polygon object
- □ throws an error with positive index
- □ calls translate after addVertex and returns Polygon object
- □ throws an error with vertices.length=2
- □ calls scale after addVertex and returns Polygon object
- □ throws an error with array vertex.x
- □ throws an error with string vector.x
- □ calls isPointInside and returns false
- □ calls calculatePerimeter after addVertex and returns positive
- □ calls removeVertex after addVertex and returns Polygon object
- □ throws an error with string factor
- □ throws an error with undefined vertex

Time Left on this task: 0:59:56



□ calls rotate and returns Polygon object

□ throws an error with vertices.length=1

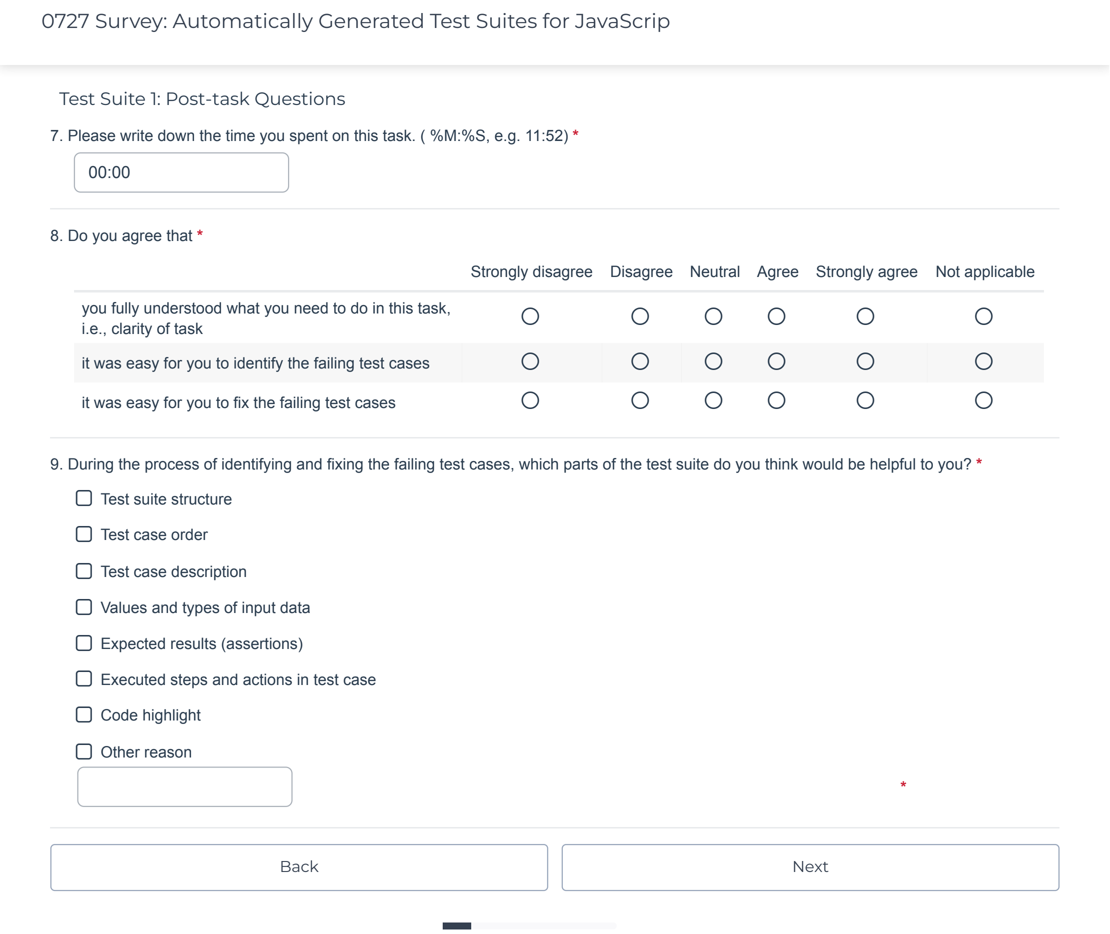
□ throws an error with undefined point.y

□ calls translate and returns Polygon object

6. For the test cases that you selected in the previous question, please explain why you think these test cases will fail. *

	test case name	reason
Bug1		
Bug2		
Bug3		
Bug4		
Bug5		

Time Left on this task: 0:59:56



0727 Survey: Automatically Generated Test Suites for JavaScrip

Test Suite 1: Post-task Questions

7. Please write down the time you spent on this task. (%M:%S, e.g. 11:52) *

00:00

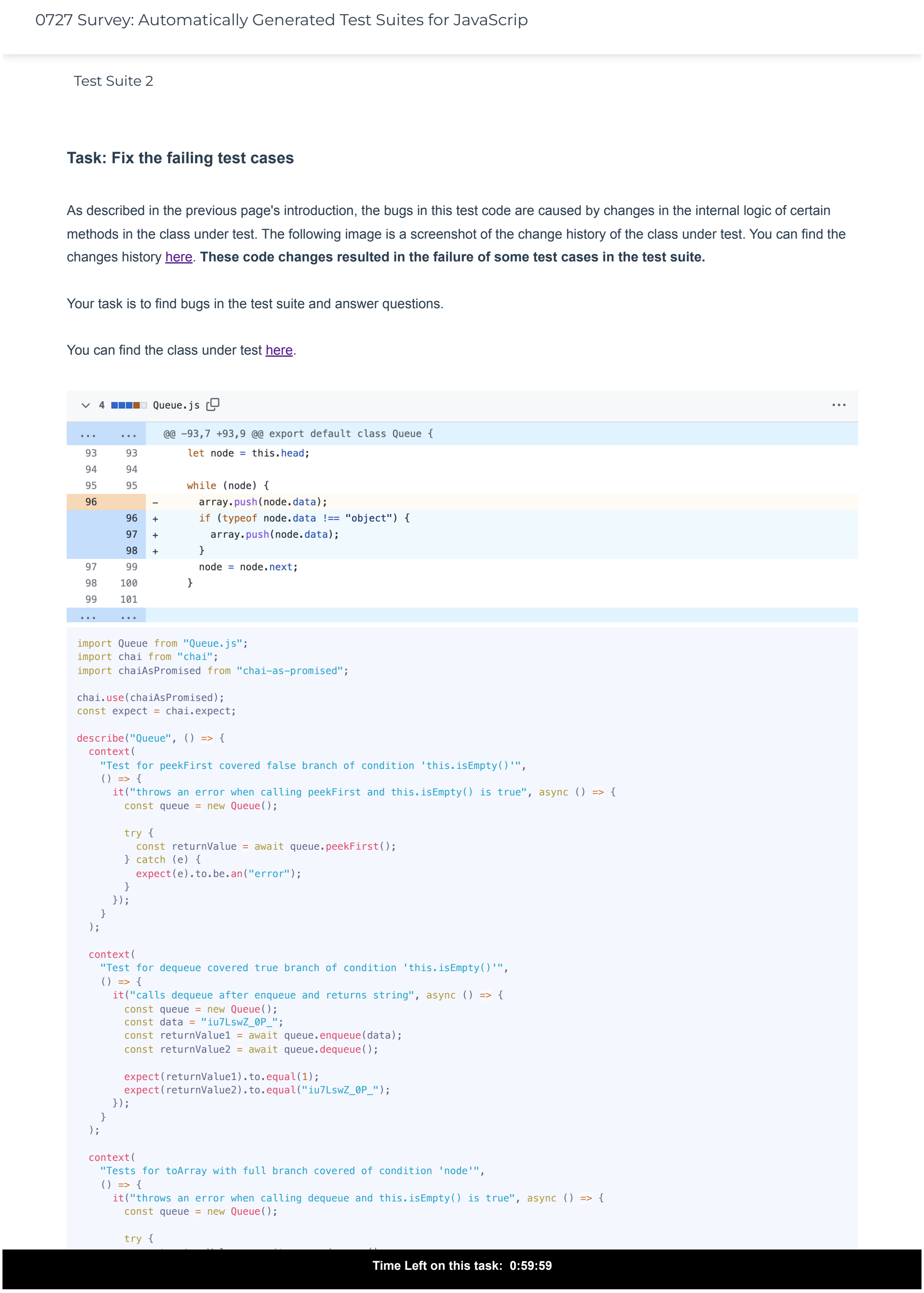
8. Do you agree that *

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree	Not applicable
you fully understood what you need to do in this task, i.e., clarity of task	○	○	○	○	○	○
it was easy for you to identify the failing test cases	○	○	○	○	○	○
it was easy for you to fix the failing test cases	○	○	○	○	○	○

9. During the process of identifying and fixing the failing test cases, which parts of the test suite do you think would be helpful to you? *

- □ Test suite structure
- □ Test case order
- □ Test case description
- □ Values and types of input data
- □ Expected results (assertions)
- □ Executed steps and actions in test case
- □ Code highlight
- □ Other reason

*



0727 Survey: Automatically Generated Test Suites for JavaScrip

Test Suite 2

Task: Fix the failing test cases

As described in the previous page's introduction, the bugs in this test code are caused by changes in the internal logic of certain methods in the class under test. The following image is a screenshot of the change history of the class under test. You can find the changes history here. **These code changes resulted in the failure of some test cases in the test suite.**

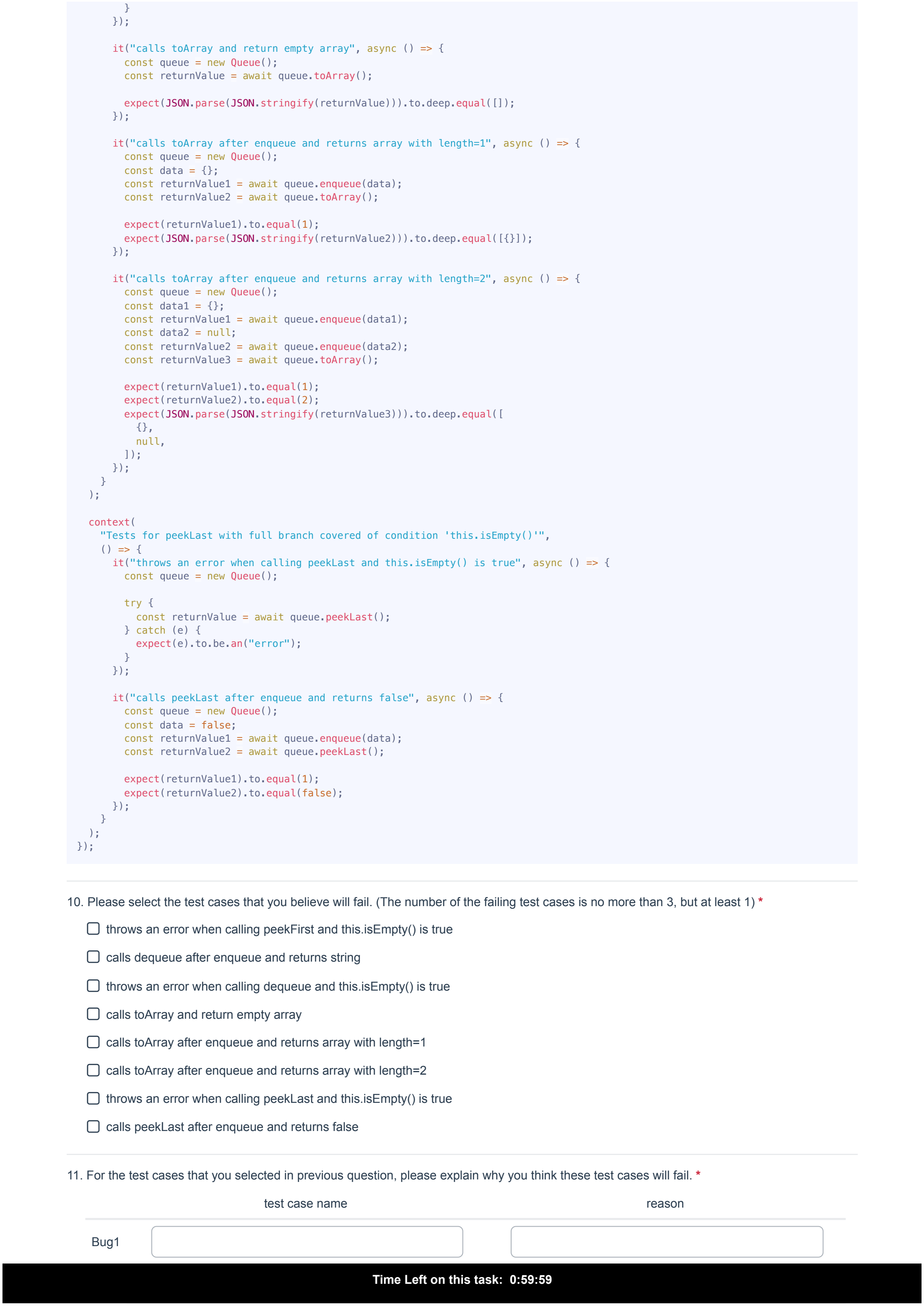
Your task is to find bugs in the test suite and answer questions.

You can find the class under test here.

```
4 Queue.js

@@ -93,7 +93,9 @@ export default class Queue {
      let node = this.head;

      while (node) {
-       array.push(node.data);
+       if (typeof node.data !== "object") {
+         array.push(node.data);
+       }
        node = node.next;
      }

```

```
import Queue from "Queue.js";
import chai from "chai";
import chaiAsPromised from "chai-as-promised";

chai.use(chaiAsPromised);
const expect = chai.expect;

describe("Queue", () => {
  context(
    "Test for peekFirst covered false branch of condition 'this.isEmpty()'",
    () => {
      it("throws an error when calling peekFirst and this.isEmpty() is true", async () => {
        const queue = new Queue();

        try {
          const returnValue = await queue.peekFirst();
        } catch (e) {
          expect(e).to.be.an("error");
        }
      });
    }
  );

  context(
    "Test for dequeue covered true branch of condition 'this.isEmpty()'",
    () => {
      it("calls dequeue after enqueue and returns string", async () => {
        const queue = new Queue();
        const data = "iu7LswZ_0P_";
        const returnValue1 = await queue.enqueue(data);
        const returnValue2 = await queue.dequeue();

        expect(returnValue1).to.equal(1);
        expect(returnValue2).to.equal("iu7LswZ_0P_");
      });
    }
  );

  context(
    "Tests for toArray with full branch covered of condition 'node'",
    () => {
      it("throws an error when calling dequeue and this.isEmpty() is true", async () => {
        const queue = new Queue();

        try {
```

Time Left on this task: 0:59:59



```
        }
      });

      it("calls toArray and return empty array", async () => {
        const queue = new Queue();
        const returnValue = await queue.toArray();

        expect(JSON.parse(JSON.stringify(returnValue))).to.deep.equal([]);
      });

      it("calls toArray after enqueue and returns array with length=1", async () => {
        const queue = new Queue();
        const data = {};
        const returnValue1 = await queue.enqueue(data);
        const returnValue2 = await queue.toArray();

        expect(returnValue1).to.equal(1);
        expect(JSON.parse(JSON.stringify(returnValue2))).to.deep.equal([{}]);
      });

      it("calls toArray after enqueue and returns array with length=2", async () => {
        const queue = new Queue();
        const data1 = {};
        const returnValue1 = await queue.enqueue(data1);
        const data2 = null;
        const returnValue2 = await queue.enqueue(data2);
        const returnValue3 = await queue.toArray();

        expect(returnValue1).to.equal(1);
        expect(returnValue2).to.equal(2);
        expect(JSON.parse(JSON.stringify(returnValue3))).to.deep.equal([
          {},
          null,
        ]);
      });
    }
  );

  context(
    "Tests for peekLast with full branch covered of condition 'this.isEmpty()'",
    () => {
      it("throws an error when calling peekLast and this.isEmpty() is true", async () => {
        const queue = new Queue();

        try {
          const returnValue = await queue.peekLast();
        } catch (e) {
          expect(e).to.be.an("error");
        }
      });

      it("calls peekLast after enqueue and returns false", async () => {
        const queue = new Queue();
        const data = false;
        const returnValue1 = await queue.enqueue(data);
        const returnValue2 = await queue.peekLast();

        expect(returnValue1).to.equal(1);
        expect(returnValue2).to.equal(false);
      });
    }
  );
});
```

10. Please select the test cases that you believe will fail. (The number of the failing test cases is no more than 3, but at least 1) *

☐ throws an error when calling peekFirst and this.isEmpty() is true

☐ calls dequeue after enqueue and returns string

☐ throws an error when calling dequeue and this.isEmpty() is true

☐ calls toArray and return empty array

☐ calls toArray after enqueue and returns array with length=1

☐ calls toArray after enqueue and returns array with length=2

☐ throws an error when calling peekLast and this.isEmpty() is true

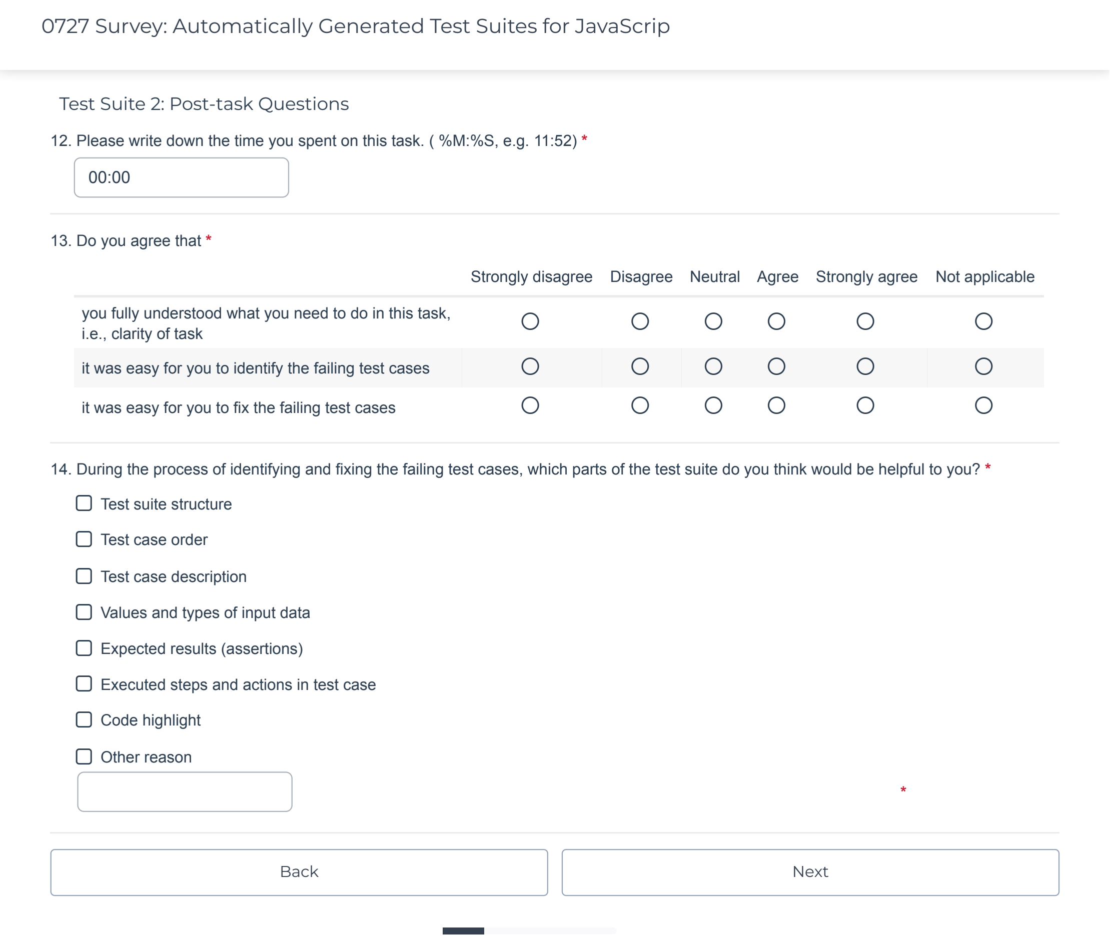
☐ calls peekLast after enqueue and returns false

11. For the test cases that you selected in previous question, please explain why you think these test cases will fail. *

	test case name	reason
Bug1		

Time Left on this task: 0:59:59

Time Left on this task: 0:59:59



0727 Survey: Automatically Generated Test Suites for JavaScrip

Test Suite 2: Post-task Questions

12. Please write down the time you spent on this task. (%M:%S, e.g. 11:52) *

00:00

13. Do you agree that *

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree	Not applicable
you fully understood what you need to do in this task, i.e., clarity of task	○	○	○	○	○	○
it was easy for you to identify the failing test cases	○	○	○	○	○	○
it was easy for you to fix the failing test cases	○	○	○	○	○	○

14. During the process of identifying and fixing the failing test cases, which parts of the test suite do you think would be helpful to you? *

□ Test suite structure

□ Test case order

□ Test case description

□ Values and types of input data

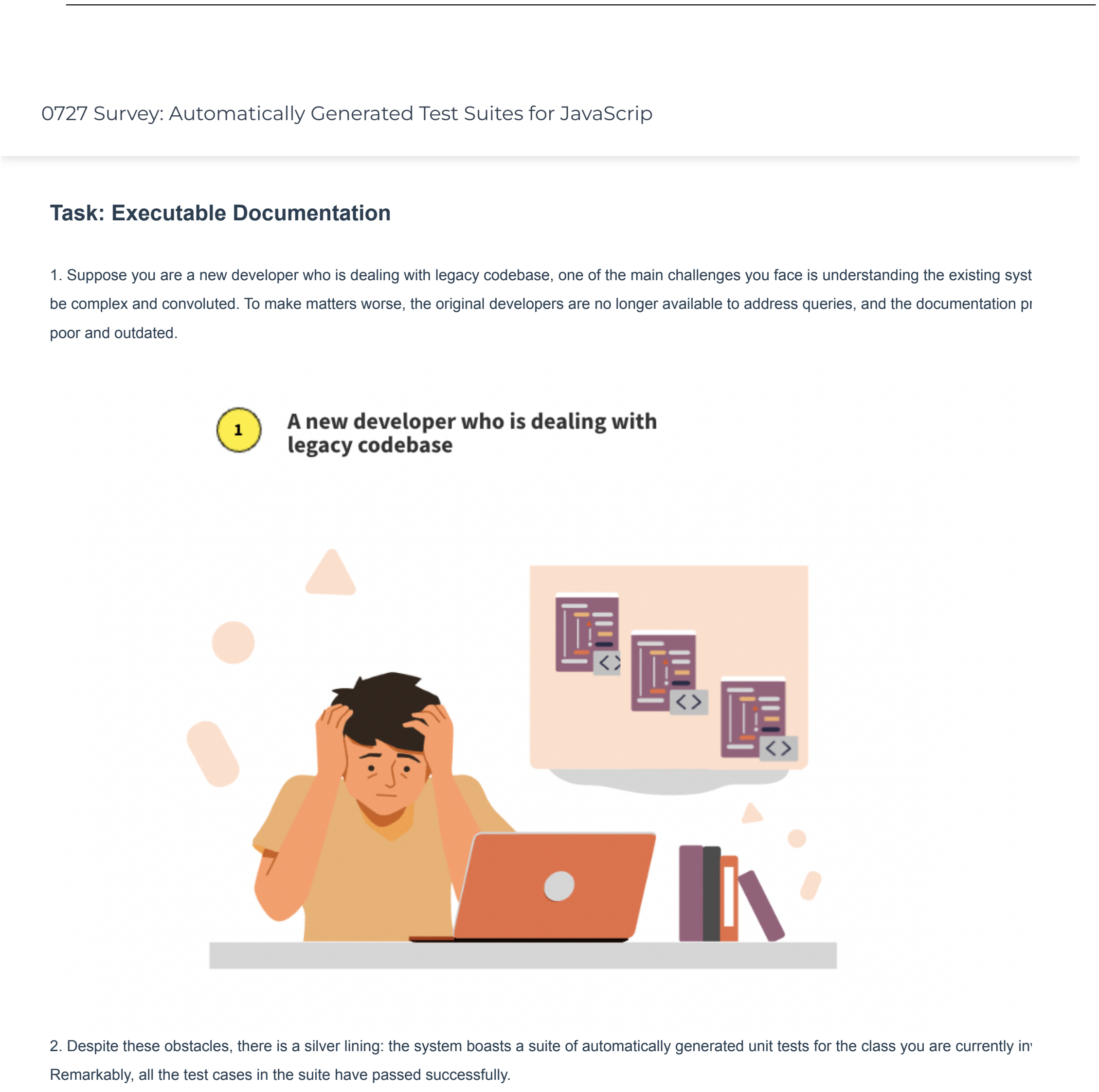
□ Expected results (assertions)

□ Executed steps and actions in test case

□ Code highlight

□ Other reason

*

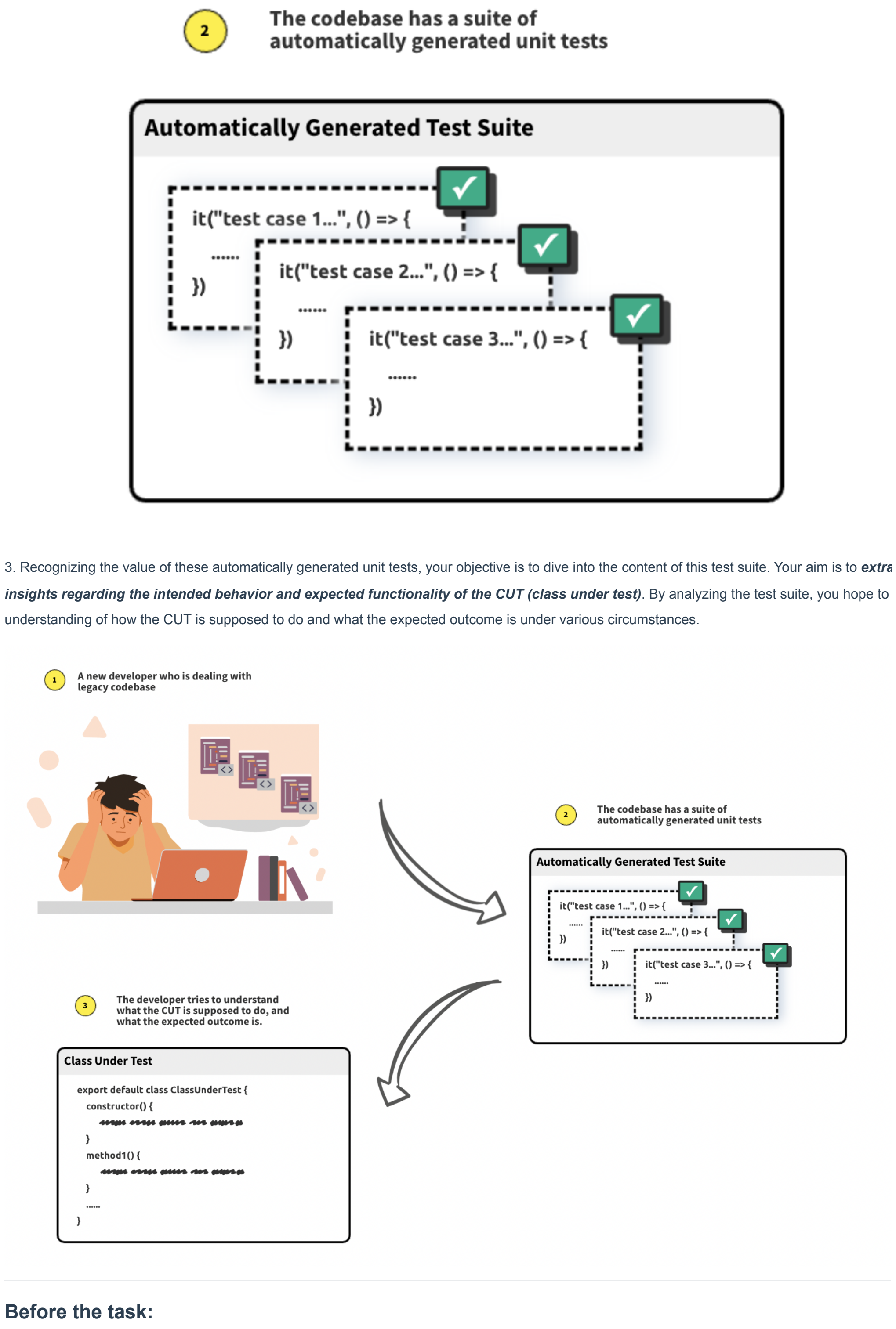


0727 Survey: Automatically Generated Test Suites for JavaScrip

Task: Executable Documentation

1. Suppose you are a new developer who is dealing with legacy codebase, one of the main challenges you face is understanding the existing syst be complex and convoluted. To make matters worse, the original developers are no longer available to address queries, and the documentation p poor and outdated.

2. Despite these obstacles, there is a silver lining: the system boasts a suite of automatically generated unit tests for the class you are currently in Remarkably, all the test cases in the suite have passed successfully.



2
The codebase has a suite of automatically generated unit tests
Automatically Generated Test Suite

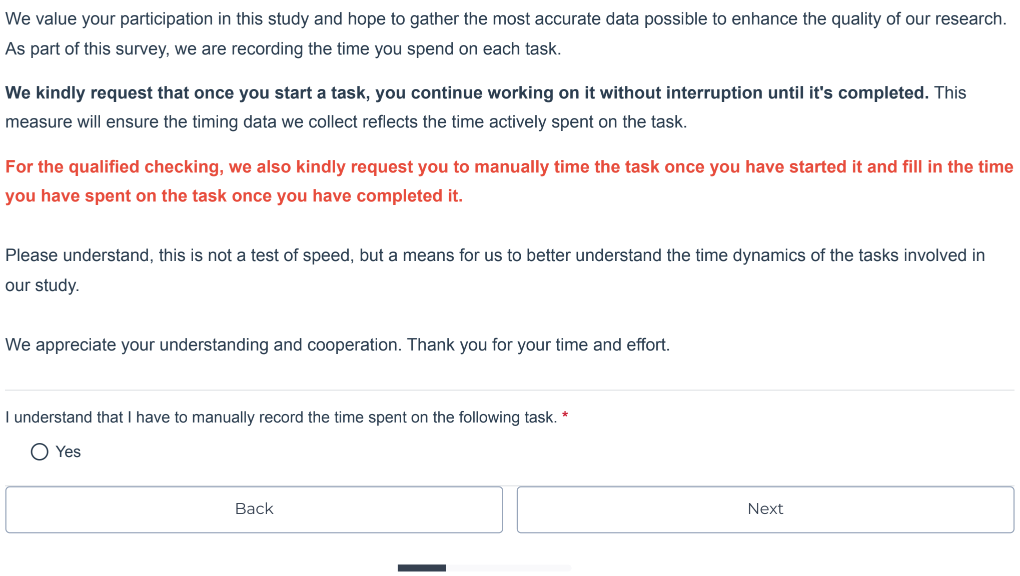
```
it("test case 1...", () => {
  ......
})
it("test case 2...", () => {
  ......
})
it("test case 3...", () => {
  ......
})
```

3. Recognizing the value of these automatically generated unit tests, your objective is to dive into the content of this test suite. Your aim is to ***extra insights regarding the intended behavior and expected functionality of the CUT (class under test)***. By analyzing the test suite, you hope to understanding of how the CUT is supposed to do and what the expected outcome is under various circumstances.

1
A new developer who is dealing with legacy codebase
2
The codebase has a suite of automatically generated unit tests
Automatically Generated Test Suite
it("test case 1...", () => {
......
})
it("test case 2...", () => {
......
})
it("test case 3...", () => {
......
})
3
The developer tries to understand what the CUT is supposed to do, and what the expected outcome is.
Class Under Test

```
export default class ClassUnderTest {
  constructor() {
  }
  method1() {
  }
  ......
}
```

Before the task:



We value your participation in this study and hope to gather the most accurate data possible to enhance the quality of our research. As part of this survey, we are recording the time you spend on each task.

We kindly request that once you start a task, you continue working on it without interruption until it's completed. This measure will ensure the timing data we collect reflects the time actively spent on the task.

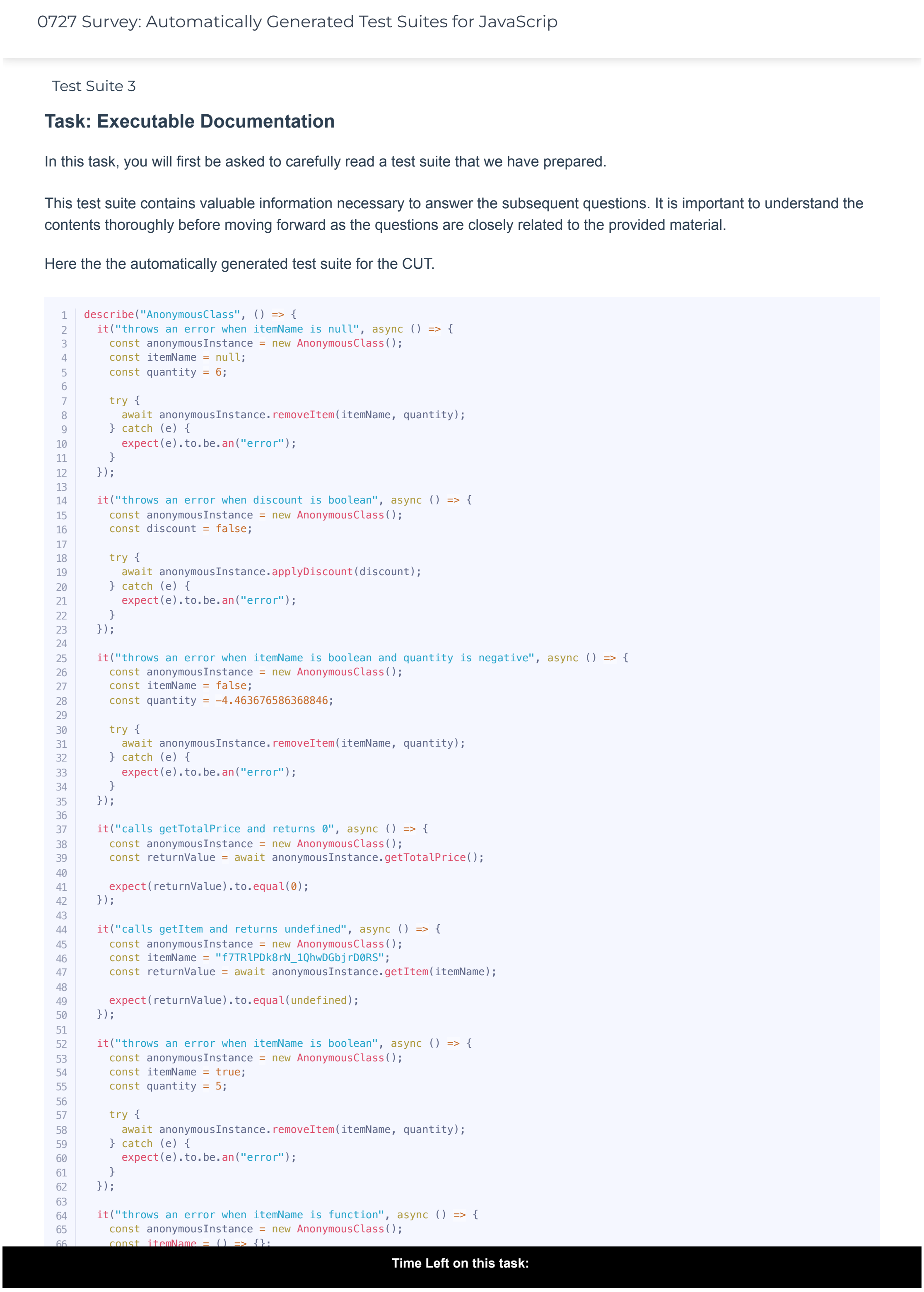
For the qualified checking, we also kindly request you to manually time the task once you have started it and fill in the time you have spent on the task once you have completed it.

Please understand, this is not a test of speed, but a means for us to better understand the time dynamics of the tasks involved in our study.

We appreciate your understanding and cooperation. Thank you for your time and effort.

I understand that I have to manually record the time spent on the following task. *

○ Yes



0727 Survey: Automatically Generated Test Suites for JavaScrip

Test Suite 3

Task: Executable Documentation

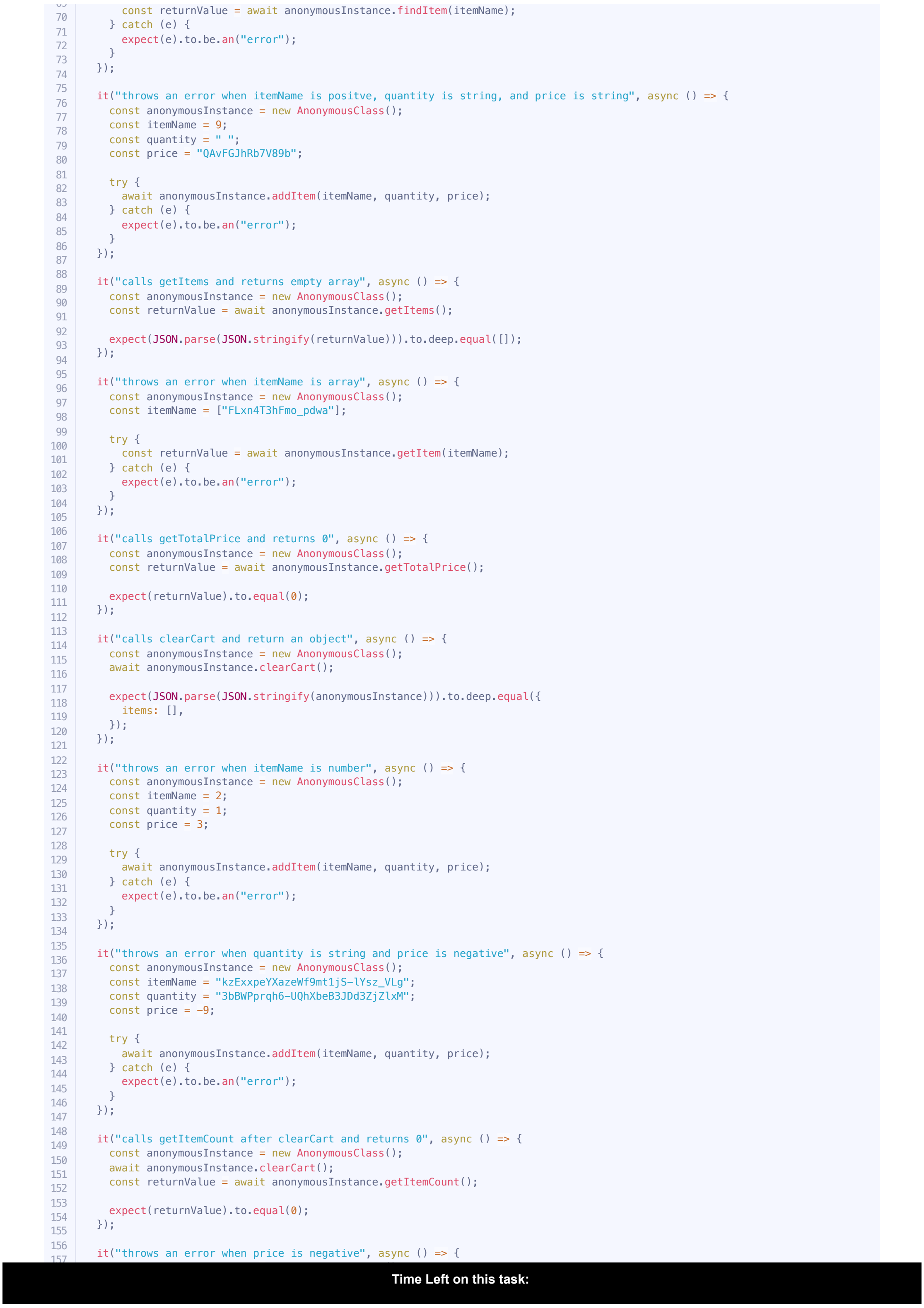
In this task, you will first be asked to carefully read a test suite that we have prepared.

This test suite contains valuable information necessary to answer the subsequent questions. It is important to understand the contents thoroughly before moving forward as the questions are closely related to the provided material.

Here the the automatically generated test suite for the CUT.

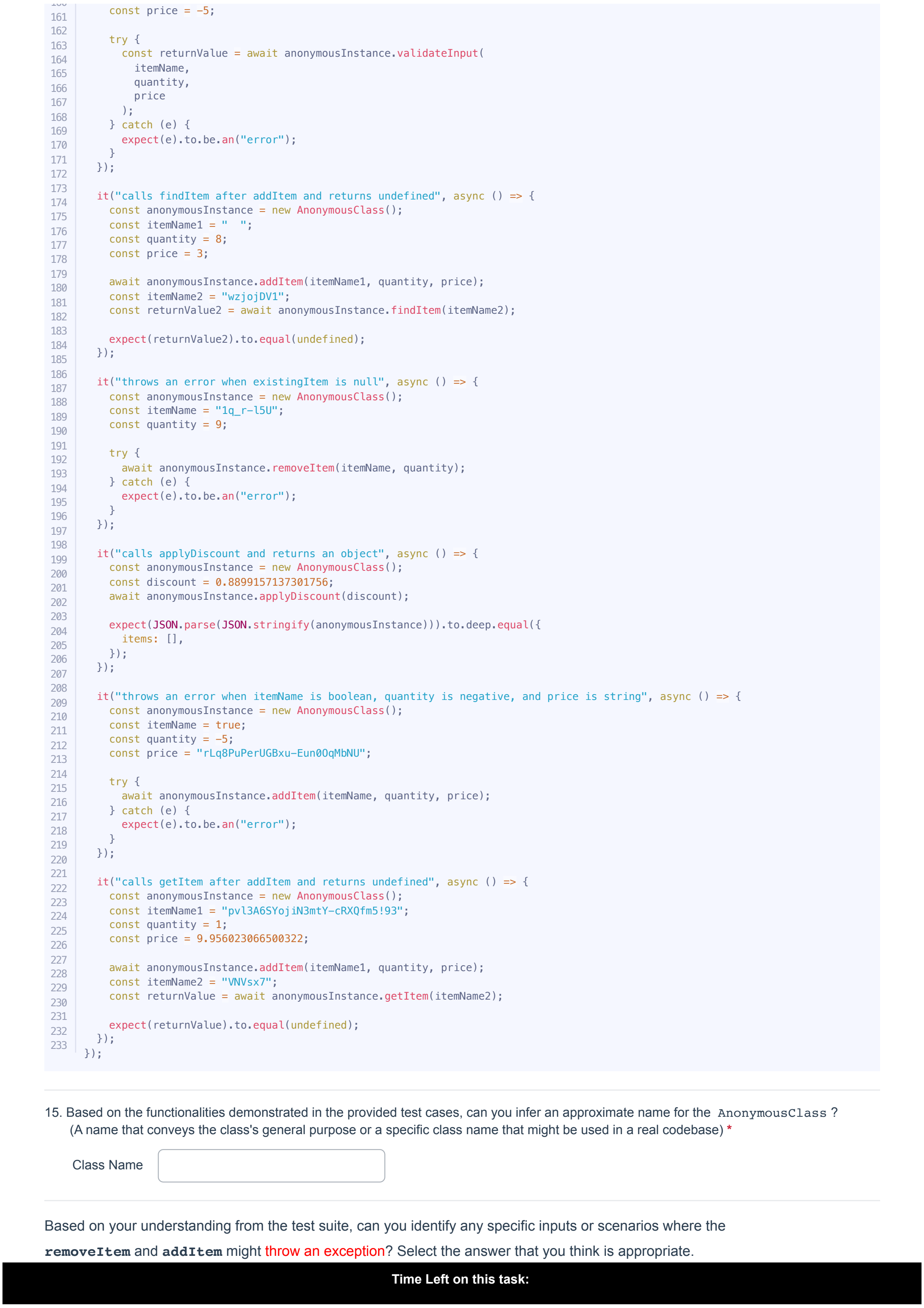
```javascript
describe("AnonymousClass", () => {
  it("throws an error when itemName is null", async () => {
    const anonymousInstance = new AnonymousClass();
    const itemName = null;
    const quantity = 6;

    try {
      await anonymousInstance.removeItem(itemName, quantity);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error when discount is boolean", async () => {
    const anonymousInstance = new AnonymousClass();
    const discount = false;

    try {
      await anonymousInstance.applyDiscount(discount);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error when itemName is boolean and quantity is negative", async () => {
    const anonymousInstance = new AnonymousClass();
    const itemName = false;
    const quantity = -4.463676586368846;

    try {
      await anonymousInstance.removeItem(itemName, quantity);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("calls getTotalPrice and returns 0", async () => {
    const anonymousInstance = new AnonymousClass();
    const returnValue = await anonymousInstance.getTotalPrice();

    expect(returnValue).to.equal(0);
  });

  it("calls getItem and returns undefined", async () => {
    const anonymousInstance = new AnonymousClass();
    const itemName = "f7TRlPDk8rN_1QhwDGbjrD0RS";
    const returnValue = await anonymousInstance.getItem(itemName);

    expect(returnValue).to.equal(undefined);
  });

  it("throws an error when itemName is boolean", async () => {
    const anonymousInstance = new AnonymousClass();
    const itemName = true;
    const quantity = 5;

    try {
      await anonymousInstance.removeItem(itemName, quantity);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error when itemName is function", async () => {
    const anonymousInstance = new AnonymousClass();
    const itemName = () => {};
```

Time Left on this task:



```
      const returnValue = await anonymousInstance.findItem(itemName);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error when itemName is positve, quantity is string, and price is string", async () => {
    const anonymousInstance = new AnonymousClass();
    const itemName = 9;
    const quantity = " ";
    const price = "QAvFGJhRb7V89b";

    try {
      await anonymousInstance.addItem(itemName, quantity, price);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("calls getItems and returns empty array", async () => {
    const anonymousInstance = new AnonymousClass();
    const returnValue = await anonymousInstance.getItems();

    expect(JSON.parse(JSON.stringify(returnValue))).to.deep.equal([]);
  });

  it("throws an error when itemName is array", async () => {
    const anonymousInstance = new AnonymousClass();
    const itemName = ["FLxn4T3hFmo_pdwa"];

    try {
      const returnValue = await anonymousInstance.getItem(itemName);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("calls getTotalPrice and returns 0", async () => {
    const anonymousInstance = new AnonymousClass();
    const returnValue = await anonymousInstance.getTotalPrice();

    expect(returnValue).to.equal(0);
  });

  it("calls clearCart and return an object", async () => {
    const anonymousInstance = new AnonymousClass();
    await anonymousInstance.clearCart();

    expect(JSON.parse(JSON.stringify(anonymousInstance))).to.deep.equal({
      items: [],
    });
  });

  it("throws an error when itemName is number", async () => {
    const anonymousInstance = new AnonymousClass();
    const itemName = 2;
    const quantity = 1;
    const price = 3;

    try {
      await anonymousInstance.addItem(itemName, quantity, price);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error when quantity is string and price is negative", async () => {
    const anonymousInstance = new AnonymousClass();
    const itemName = "kzExxpeYXazeWf9mt1jS-lYsz_VLg";
    const quantity = "3bBWPprqh6-UQhXbeB3JDd3ZjZlxM";
    const price = -9;

    try {
      await anonymousInstance.addItem(itemName, quantity, price);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("calls getItemCount after clearCart and returns 0", async () => {
    const anonymousInstance = new AnonymousClass();
    await anonymousInstance.clearCart();
    const returnValue = await anonymousInstance.getItemCount();

    expect(returnValue).to.equal(0);
  });

  it("throws an error when price is negative", async () => {
```

Time Left on this task:



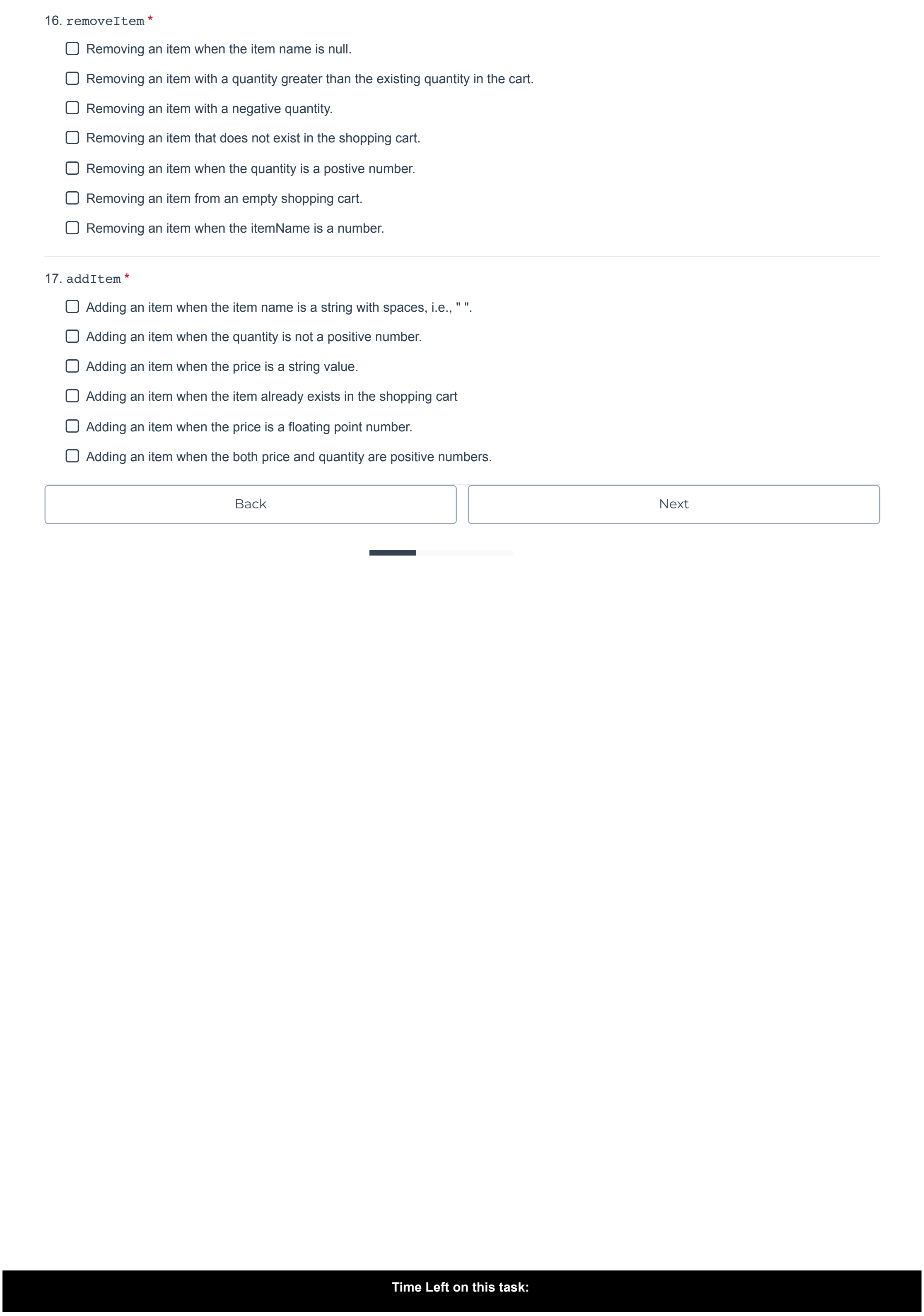
```
    const price = -5;

    try {
      const returnValue = await anonymousInstance.validateInput(
        itemName,
        quantity,
        price
      );
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("calls findItem after addItem and returns undefined", async () => {
    const anonymousInstance = new AnonymousClass();
    const itemName1 = "  ";
    const quantity = 8;
    const price = 3;

    await anonymousInstance.addItem(itemName1, quantity, price);
    const itemName2 = "wzjojDV1";
    const returnValue2 = await anonymousInstance.findItem(itemName2);

    expect(returnValue2).to.equal(undefined);
  });

  it("throws an error when existingItem is null", async () => {
    const anonymousInstance = new AnonymousClass();
    const itemName = "1q_r-l5U";
    const quantity = 9;

    try {
      await anonymousInstance.removeItem(itemName, quantity);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("calls applyDiscount and returns an object", async () => {
    const anonymousInstance = new AnonymousClass();
    const discount = 0.8899157137301756;
    await anonymousInstance.applyDiscount(discount);

    expect(JSON.parse(JSON.stringify(anonymousInstance))).to.deep.equal({
      items: [],
    });
  });

  it("throws an error when itemName is boolean, quantity is negative, and price is string", async () => {
    const anonymousInstance = new AnonymousClass();
    const itemName = true;
    const quantity = -5;
    const price = "rLq8PuPerUGBxu-Eun0OqMbNU";

    try {
      await anonymousInstance.addItem(itemName, quantity, price);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("calls getItem after addItem and returns undefined", async () => {
    const anonymousInstance = new AnonymousClass();
    const itemName1 = "pvl3A6SYojiN3mtY-cRXQfm5!93";
    const quantity = 1;
    const price = 9.956023066500322;

    await anonymousInstance.addItem(itemName1, quantity, price);
    const itemName2 = "VNVsx7";
    const returnValue = await anonymousInstance.getItem(itemName2);

    expect(returnValue).to.equal(undefined);
  });
});
```

15. Based on the functionalities demonstrated in the provided test cases, can you infer an approximate name for the `AnonymousClass` ?
(A name that conveys the class's general purpose or a specific class name that might be used in a real codebase) *

Class Name

Based on your understanding from the test suite, can you identify any specific inputs or scenarios where the **`removeItem`** and **`addItem`** might throw an exception? Select the answer that you think is appropriate.

Time Left on this task:



16. `removeItem` *

□ Removing an item when the item name is null.

□ Removing an item with a quantity greater than the existing quantity in the cart.

□ Removing an item with a negative quantity.

□ Removing an item that does not exist in the shopping cart.

□ Removing an item when the quantity is a postive number.

□ Removing an item from an empty shopping cart.

□ Removing an item when the itemName is a number.

17. `addItem` *

□ Adding an item when the item name is a string with spaces, i.e., " ".

□ Adding an item when the quantity is not a positive number.

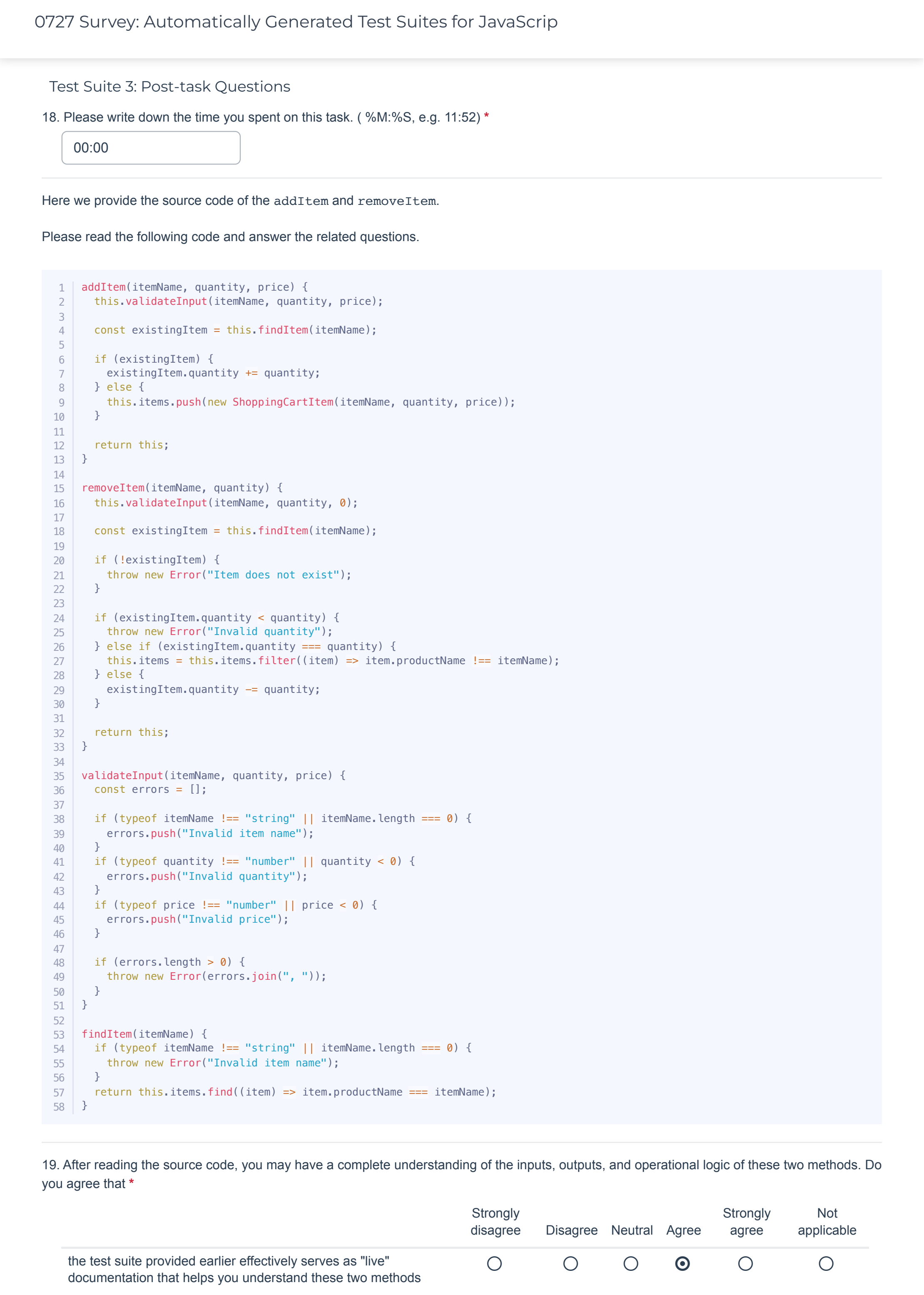
□ Adding an item when the price is a string value.

□ Adding an item when the item already exists in the shopping cart

□ Adding an item when the price is a floating point number.

□ Adding an item when the both price and quantity are positive numbers.

Time Left on this task:



0727 Survey: Automatically Generated Test Suites for JavaScrip

Test Suite 3: Post-task Questions

18. Please write down the time you spent on this task. (%M:%S, e.g. 11:52) *

00:00

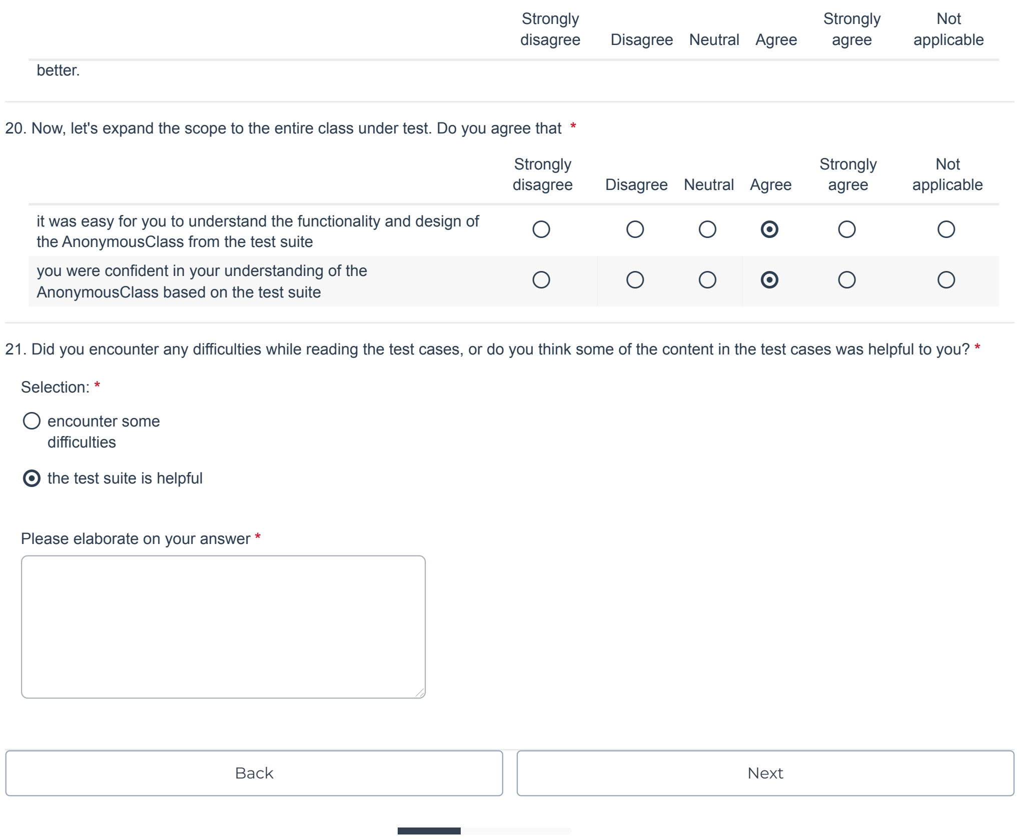
Here we provide the source code of the `addItem` and `removeItem`.

Please read the following code and answer the related questions.

```
addItem(itemName, quantity, price) {
  this.validateInput(itemName, quantity, price);

  const existingItem = this.findItem(itemName);

  if (existingItem) {
    existingItem.quantity += quantity;
  } else {
    this.items.push(new ShoppingCartItem(itemName, quantity, price));
  }

  return this;
}

removeItem(itemName, quantity) {
  this.validateInput(itemName, quantity, 0);

  const existingItem = this.findItem(itemName);

  if (!existingItem) {
    throw new Error("Item does not exist");
  }

  if (existingItem.quantity < quantity) {
    throw new Error("Invalid quantity");
  } else if (existingItem.quantity === quantity) {
    this.items = this.items.filter((item) => item.productName !== itemName);
  } else {
    existingItem.quantity -= quantity;
  }

  return this;
}

validateInput(itemName, quantity, price) {
  const errors = [];

  if (typeof itemName !== "string" || itemName.length === 0) {
    errors.push("Invalid item name");
  }
  if (typeof quantity !== "number" || quantity < 0) {
    errors.push("Invalid quantity");
  }
  if (typeof price !== "number" || price < 0) {
    errors.push("Invalid price");
  }

  if (errors.length > 0) {
    throw new Error(errors.join(", "));
  }
}

findItem(itemName) {
  if (typeof itemName !== "string" || itemName.length === 0) {
    throw new Error("Invalid item name");
  }
  return this.items.find((item) => item.productName === itemName);
}
```

19. After reading the source code, you may have a complete understanding of the inputs, outputs, and operational logic of these two methods. Do you agree that *

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree	Not applicable
the test suite provided earlier effectively serves as "live" documentation that helps you understand these two methods	○	○	○	●	○	○



	Strongly disagree	Disagree	Neutral	Agree	Strongly agree	Not applicable
better.						

20. Now, let's expand the scope to the entire class under test. Do you agree that *

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree	Not applicable
it was easy for you to understand the functionality and design of the AnonymousClass from the test suite	○	○	○	●	○	○
you were confident in your understanding of the AnonymousClass based on the test suite	○	○	○	●	○	○

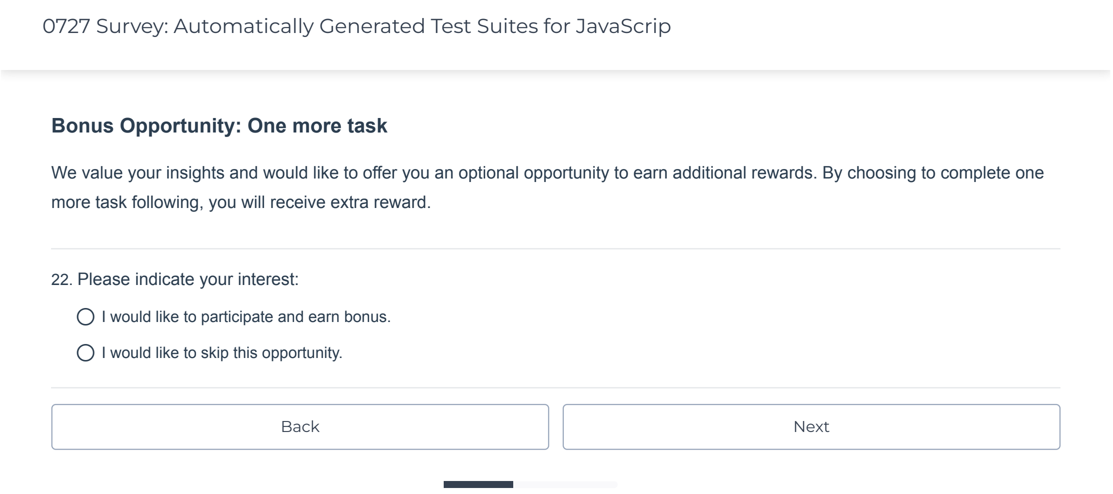
21. Did you encounter any difficulties while reading the test cases, or do you think some of the content in the test cases was helpful to you? *

Selection: *

○ encounter some difficulties

● the test suite is helpful

Please elaborate on your answer *



0727 Survey: Automatically Generated Test Suites for JavaScrip

Bonus Opportunity: One more task

We value your insights and would like to offer you an optional opportunity to earn additional rewards. By choosing to complete one more task following, you will receive extra reward.

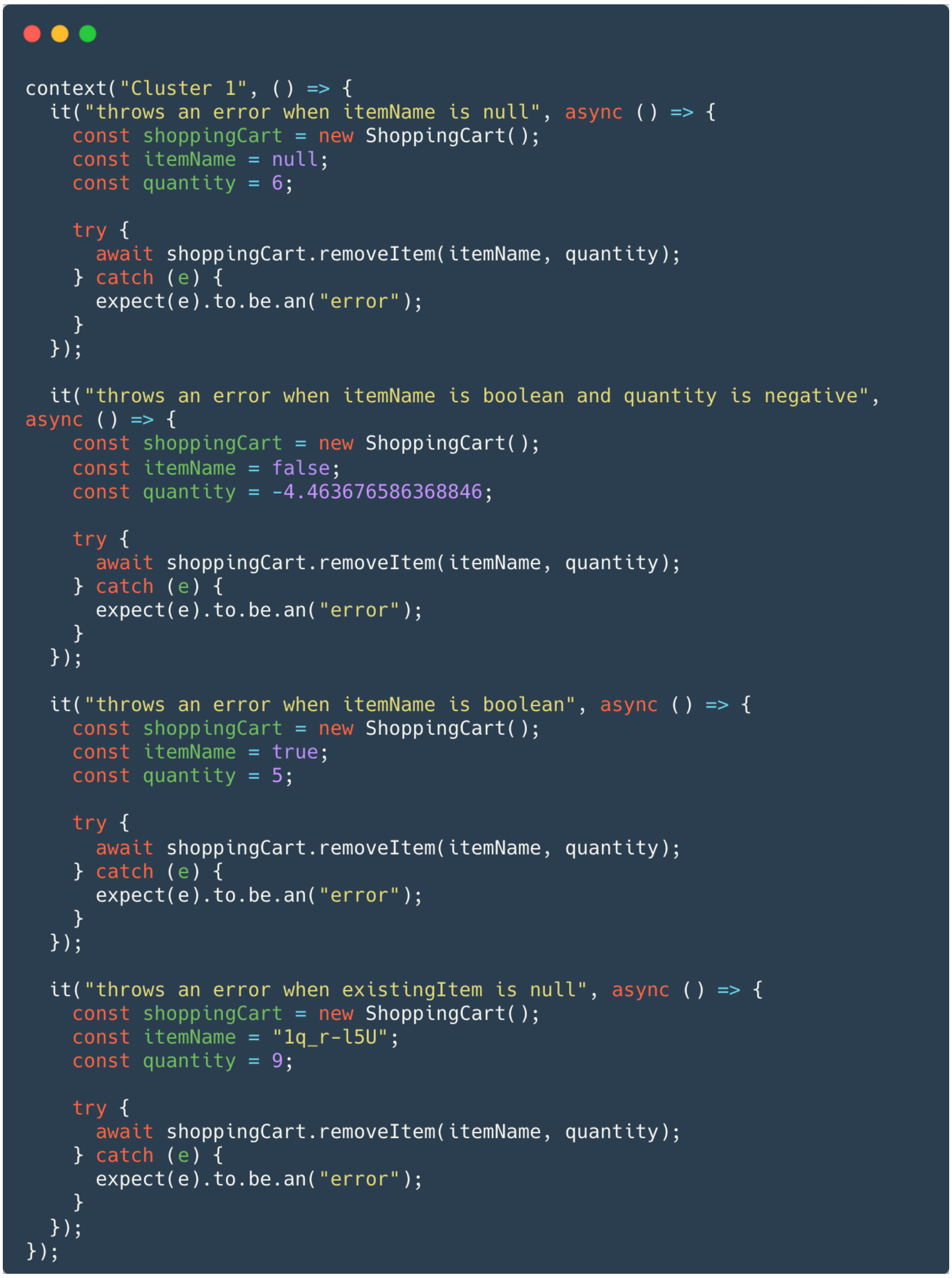
22. Please indicate your interest:

○ I would like to participate and earn bonus.

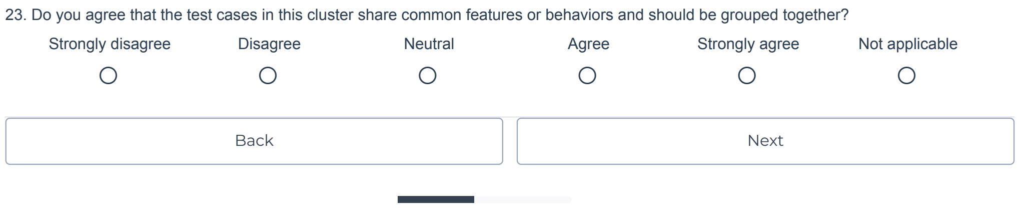
○ I would like to skip this opportunity.

0727 Survey: Automatically Generated Test Suites for JavaScrip

Cluster 1

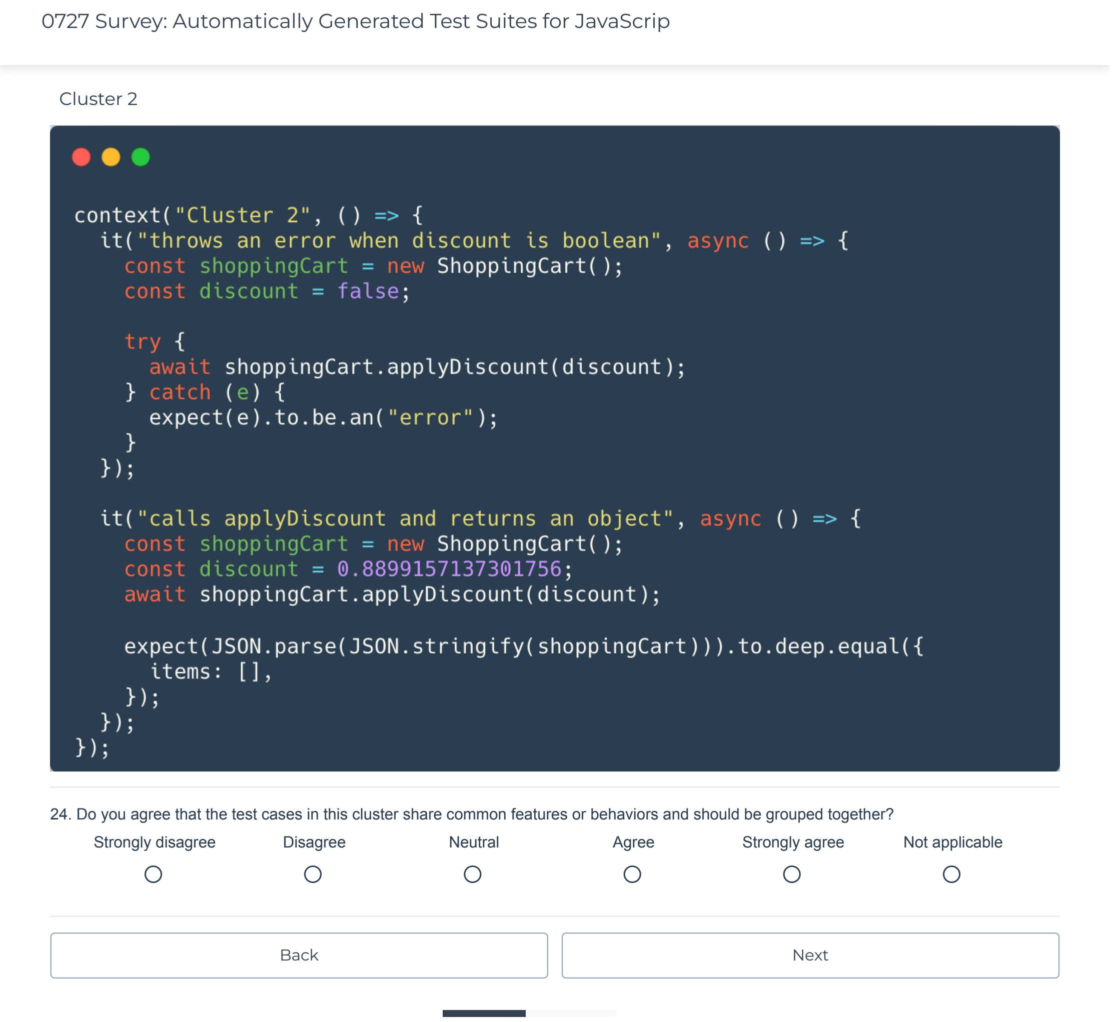


```
context("Cluster 1", () => {
  it("throws an error when itemName is null", async () => {
    const shoppingCart = new ShoppingCart();
    const itemName = null;
    const quantity = 6;

    try {
      await shoppingCart.removeItem(itemName, quantity);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error when itemName is boolean and quantity is negative",
async () => {
    const shoppingCart = new ShoppingCart();
    const itemName = false;
    const quantity = -4.463676586368846;

    try {
      await shoppingCart.removeItem(itemName, quantity);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error when itemName is boolean", async () => {
    const shoppingCart = new ShoppingCart();
    const itemName = true;
    const quantity = 5;

    try {
      await shoppingCart.removeItem(itemName, quantity);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error when existingItem is null", async () => {
    const shoppingCart = new ShoppingCart();
    const itemName = "1q_r-l5U";
    const quantity = 9;

    try {
      await shoppingCart.removeItem(itemName, quantity);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });
});
```



23. Do you agree that the test cases in this cluster share common features or behaviors and should be grouped together?

Strongly disagree	Disagree	Neutral	Agree	Strongly agree	Not applicable
○	○	○	○	○	○



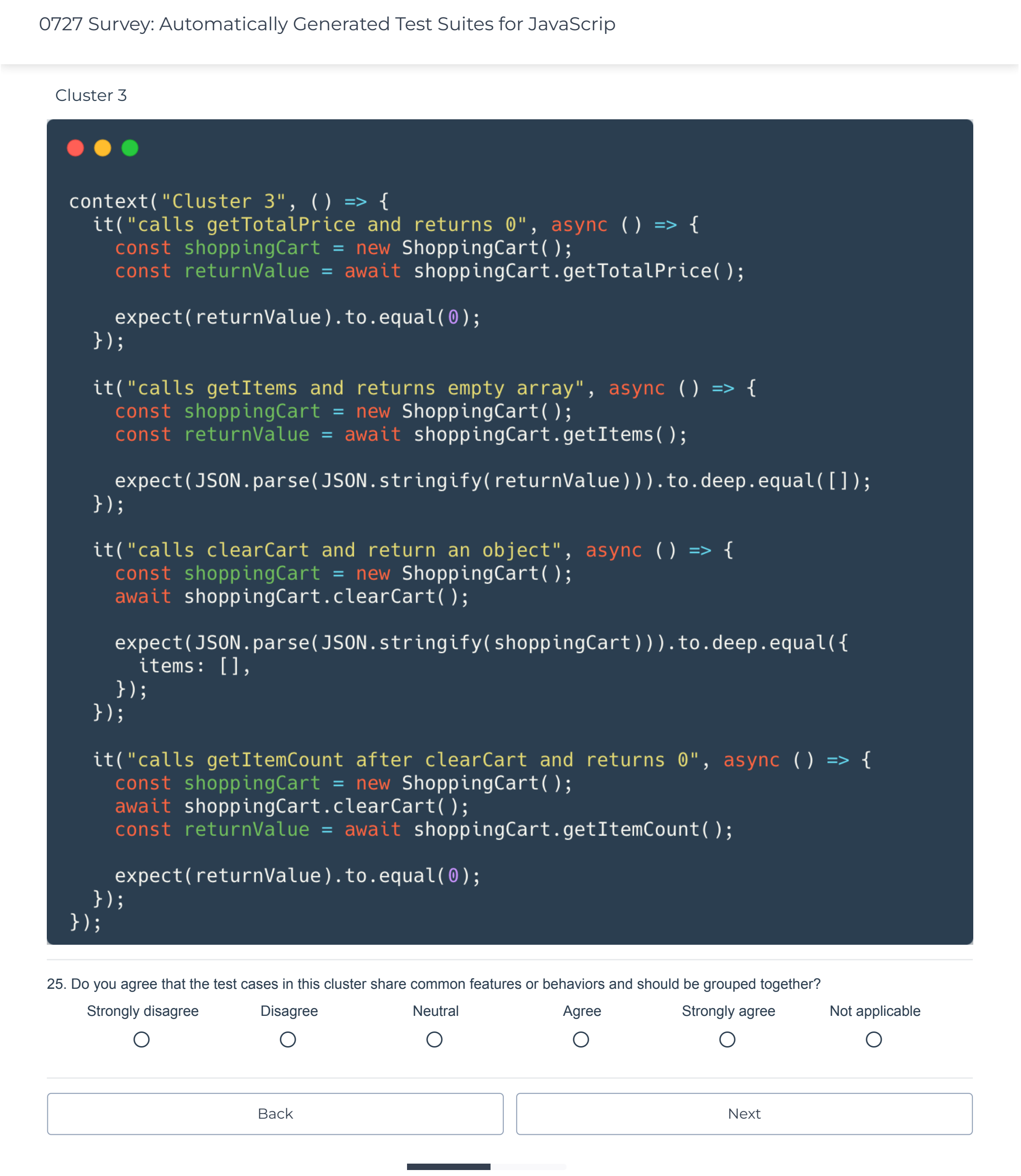
0727 Survey: Automatically Generated Test Suites for JavaScrip

Cluster 2

```
context("Cluster 2", () => {
  it("throws an error when discount is boolean", async () => {
    const shoppingCart = new ShoppingCart();
    const discount = false;

    try {
      await shoppingCart.applyDiscount(discount);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("calls applyDiscount and returns an object", async () => {
    const shoppingCart = new ShoppingCart();
    const discount = 0.8899157137301756;
    await shoppingCart.applyDiscount(discount);

    expect(JSON.parse(JSON.stringify(shoppingCart))).to.deep.equal({
      items: [],
    });
  });
});
```

24. Do you agree that the test cases in this cluster share common features or behaviors and should be grouped together?

Strongly disagree	Disagree	Neutral	Agree	Strongly agree	Not applicable
○	○	○	○	○	○



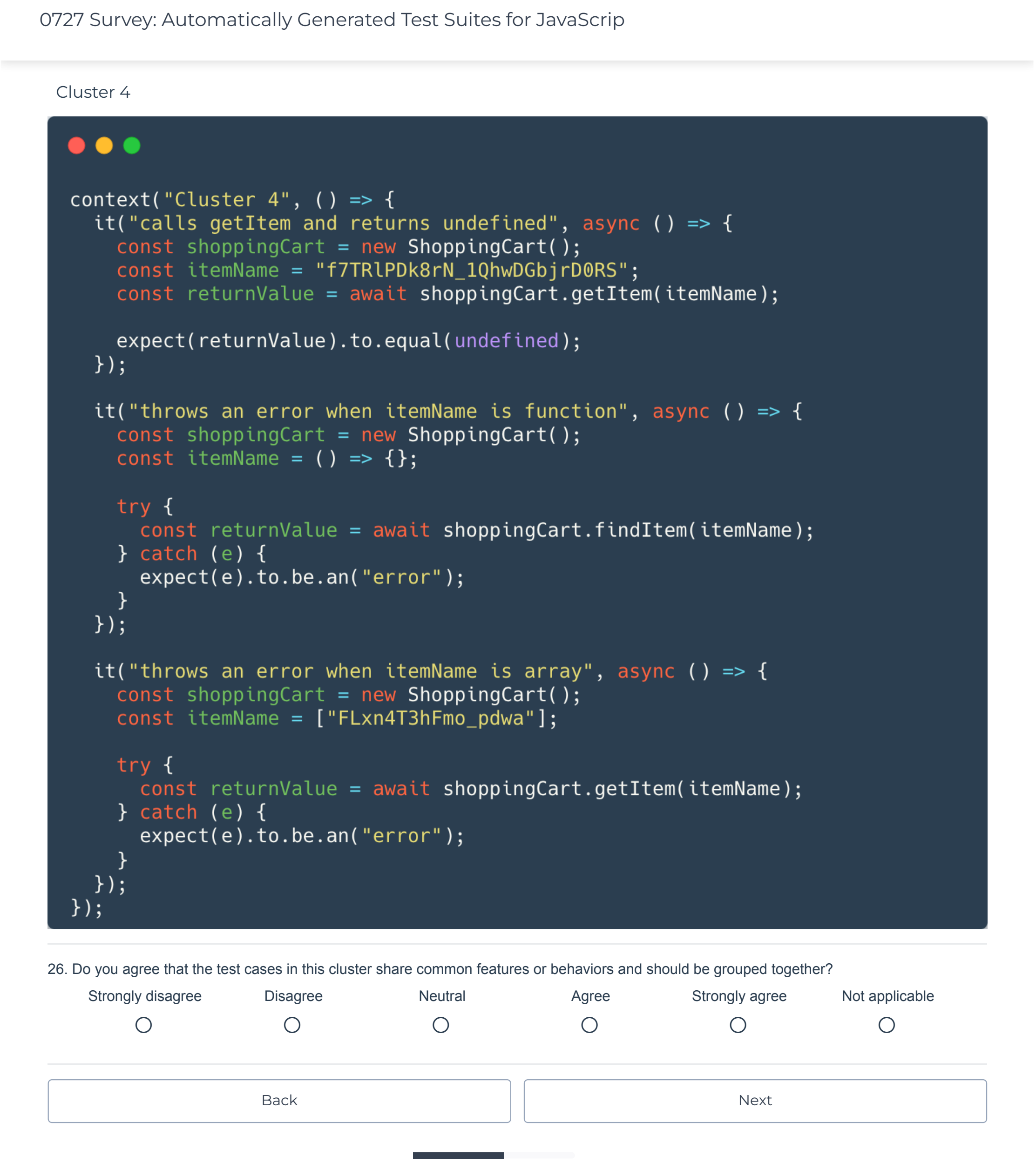
0727 Survey: Automatically Generated Test Suites for JavaScrip

Cluster 3

```
context("Cluster 3", () => {
  it("calls getTotalPrice and returns 0", async () => {
    const shoppingCart = new ShoppingCart();
    const returnValue = await shoppingCart.getTotalPrice();

    expect(returnValue).to.equal(0);
  });

  it("calls getItems and returns empty array", async () => {
    const shoppingCart = new ShoppingCart();
    const returnValue = await shoppingCart.getItems();

    expect(JSON.parse(JSON.stringify(returnValue))).to.deep.equal([]);
  });

  it("calls clearCart and return an object", async () => {
    const shoppingCart = new ShoppingCart();
    await shoppingCart.clearCart();

    expect(JSON.parse(JSON.stringify(shoppingCart))).to.deep.equal({
      items: [],
    });
  });

  it("calls getItemCount after clearCart and returns 0", async () => {
    const shoppingCart = new ShoppingCart();
    await shoppingCart.clearCart();
    const returnValue = await shoppingCart.getItemCount();

    expect(returnValue).to.equal(0);
  });
});
```

25. Do you agree that the test cases in this cluster share common features or behaviors and should be grouped together?

Strongly disagree	Disagree	Neutral	Agree	Strongly agree	Not applicable
○	○	○	○	○	○



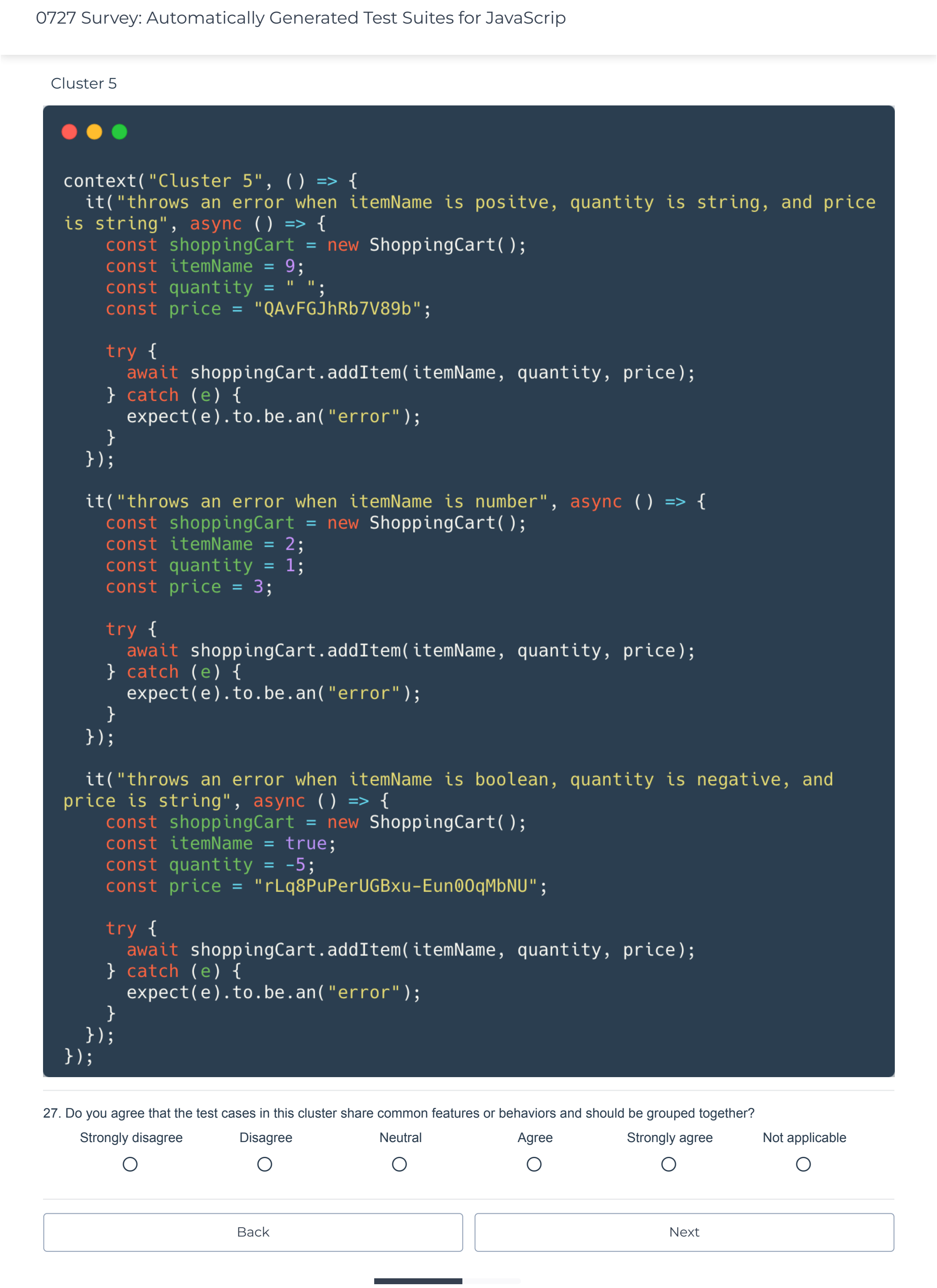
0727 Survey: Automatically Generated Test Suites for JavaScrip

Cluster 4

```
context("Cluster 4", () => {
  it("calls getItem and returns undefined", async () => {
    const shoppingCart = new ShoppingCart();
    const itemName = "f7TRlPDk8rN_1QhwDGbjrD0RS";
    const returnValue = await shoppingCart.getItem(itemName);

    expect(returnValue).to.equal(undefined);
  });

  it("throws an error when itemName is function", async () => {
    const shoppingCart = new ShoppingCart();
    const itemName = () => {};

    try {
      const returnValue = await shoppingCart.findItem(itemName);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error when itemName is array", async () => {
    const shoppingCart = new ShoppingCart();
    const itemName = ["FLxn4T3hFmo_pdwa"];

    try {
      const returnValue = await shoppingCart.getItem(itemName);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });
});
```

26. Do you agree that the test cases in this cluster share common features or behaviors and should be grouped together?

Strongly disagree	Disagree	Neutral	Agree	Strongly agree	Not applicable
○	○	○	○	○	○



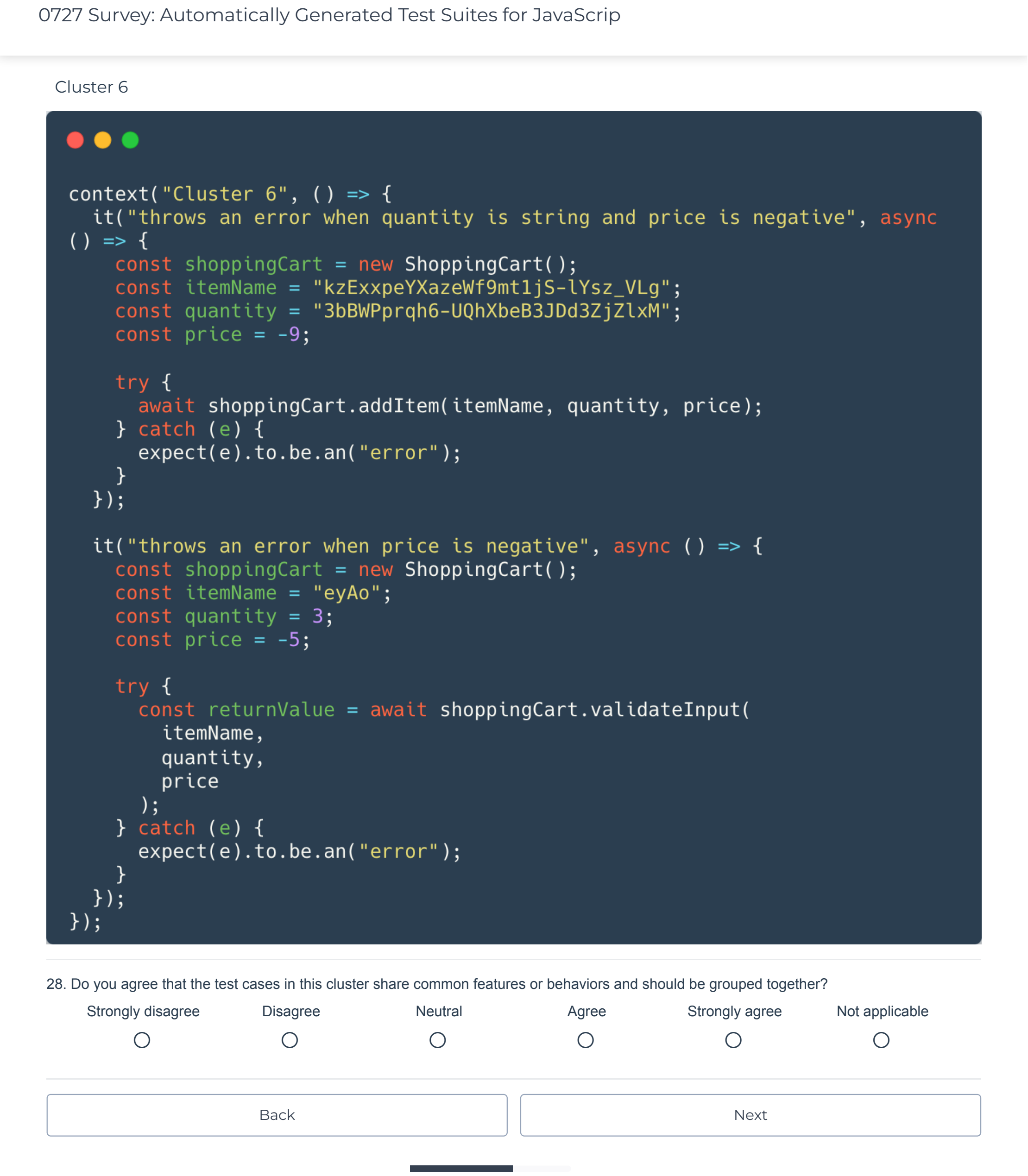
0727 Survey: Automatically Generated Test Suites for JavaScrip

Cluster 5

```
context("Cluster 5", () => {
  it("throws an error when itemName is positve, quantity is string, and price
is string", async () => {
    const shoppingCart = new ShoppingCart();
    const itemName = 9;
    const quantity = " ";
    const price = "QAvFGJhRb7V89b";

    try {
      await shoppingCart.addItem(itemName, quantity, price);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error when itemName is number", async () => {
    const shoppingCart = new ShoppingCart();
    const itemName = 2;
    const quantity = 1;
    const price = 3;

    try {
      await shoppingCart.addItem(itemName, quantity, price);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error when itemName is boolean, quantity is negative, and
price is string", async () => {
    const shoppingCart = new ShoppingCart();
    const itemName = true;
    const quantity = -5;
    const price = "rLq8PuPerUGBxu-Eun0OqMbNU";

    try {
      await shoppingCart.addItem(itemName, quantity, price);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });
});
```

27. Do you agree that the test cases in this cluster share common features or behaviors and should be grouped together?

Strongly disagree	Disagree	Neutral	Agree	Strongly agree	Not applicable
○	○	○	○	○	○



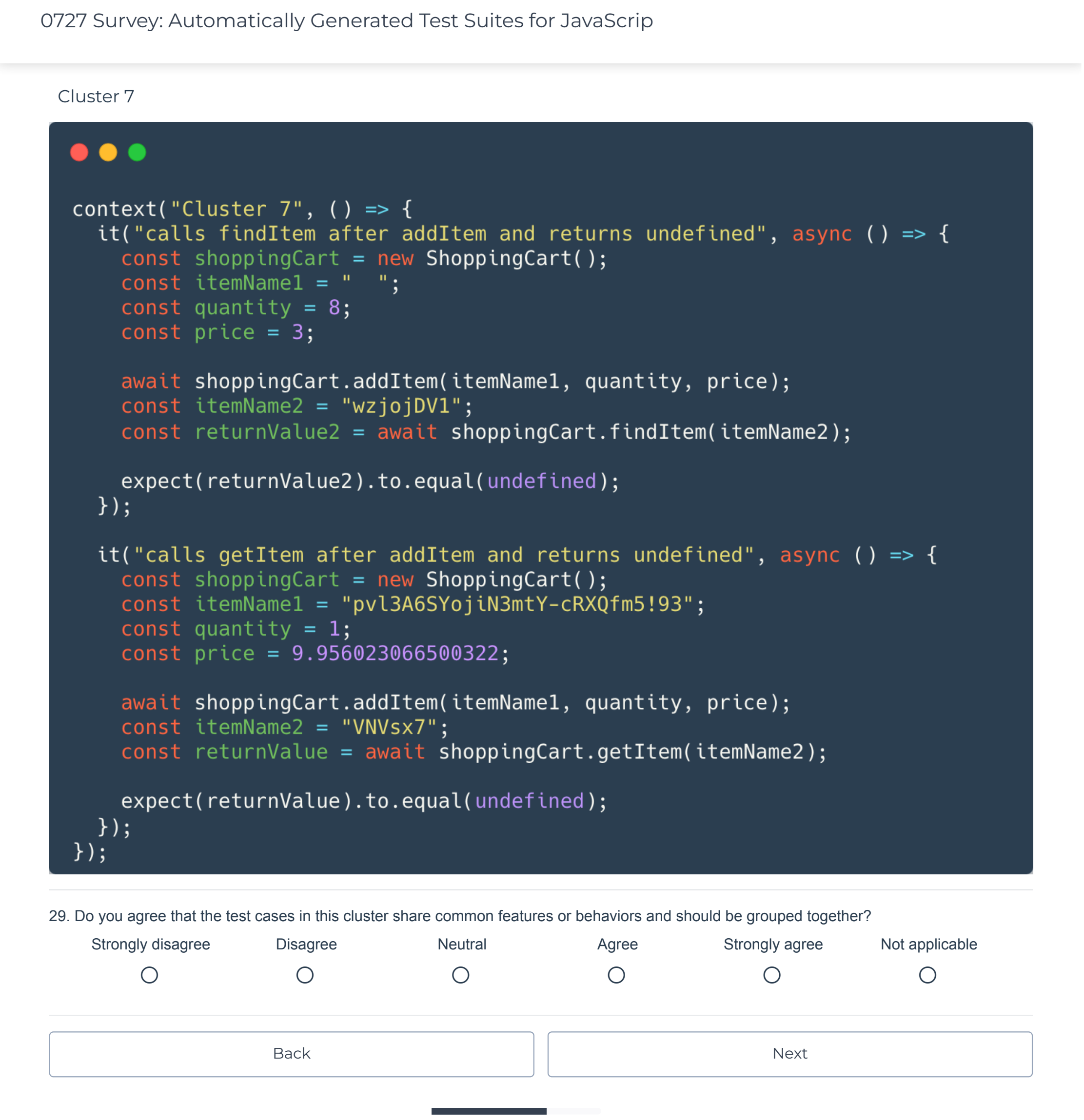
0727 Survey: Automatically Generated Test Suites for JavaScrip

Cluster 6

```
context("Cluster 6", () => {
  it("throws an error when quantity is string and price is negative", async
() => {
    const shoppingCart = new ShoppingCart();
    const itemName = "kzExxpeYXazeWf9mt1jS-lYsz_VLg";
    const quantity = "3bBWPprqh6-UQhXbeB3JDd3ZjZlxM";
    const price = -9;

    try {
      await shoppingCart.addItem(itemName, quantity, price);
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });

  it("throws an error when price is negative", async () => {
    const shoppingCart = new ShoppingCart();
    const itemName = "eyAo";
    const quantity = 3;
    const price = -5;

    try {
      const returnValue = await shoppingCart.validateInput(
        itemName,
        quantity,
        price
      );
    } catch (e) {
      expect(e).to.be.an("error");
    }
  });
});
```

28. Do you agree that the test cases in this cluster share common features or behaviors and should be grouped together?

Strongly disagree	Disagree	Neutral	Agree	Strongly agree	Not applicable
○	○	○	○	○	○



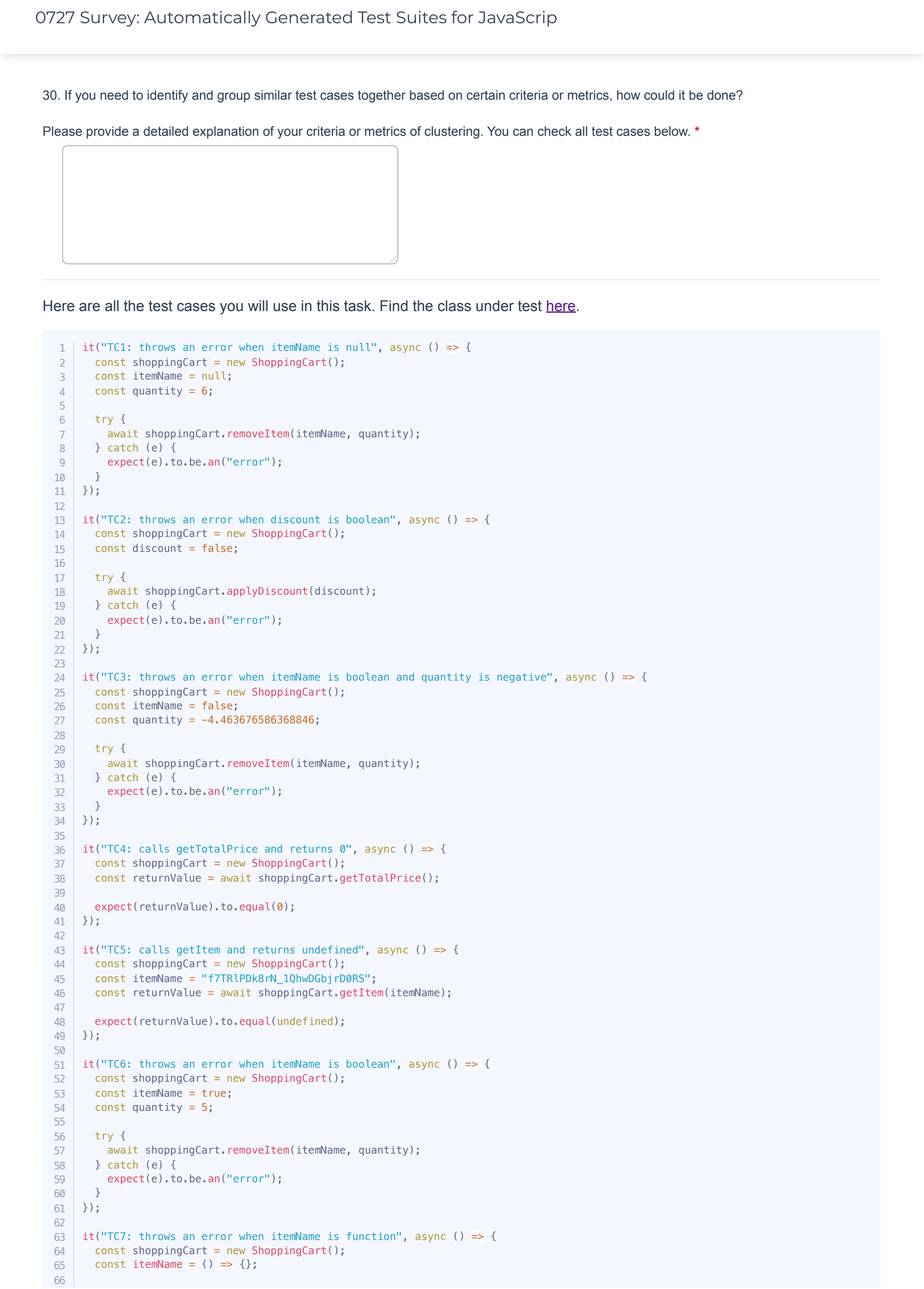
0727 Survey: Automatically Generated Test Suites for JavaScrip

Cluster 7

```
context("Cluster 7", () => {
  it("calls findItem after addItem and returns undefined", async () => {
    const shoppingCart = new ShoppingCart();
    const itemName1 = "  ";
    const quantity = 8;
    const price = 3;

    await shoppingCart.addItem(itemName1, quantity, price);
    const itemName2 = "wzjojDV1";
    const returnValue2 = await shoppingCart.findItem(itemName2);

    expect(returnValue2).to.equal(undefined);
  });

  it("calls getItem after addItem and returns undefined", async () => {
    const shoppingCart = new ShoppingCart();
    const itemName1 = "pvl3A6SYojiN3mtY-cRXQfm5!93";
    const quantity = 1;
    const price = 9.956023066500322;

    await shoppingCart.addItem(itemName1, quantity, price);
    const itemName2 = "VNVsx7";
    const returnValue = await shoppingCart.getItem(itemName2);

    expect(returnValue).to.equal(undefined);
  });
});
```

29. Do you agree that the test cases in this cluster share common features or behaviors and should be grouped together?

Strongly disagree	Disagree	Neutral	Agree	Strongly agree	Not applicable
○	○	○	○	○	○



0727 Survey: Automatically Generated Test Suites for JavaScrip

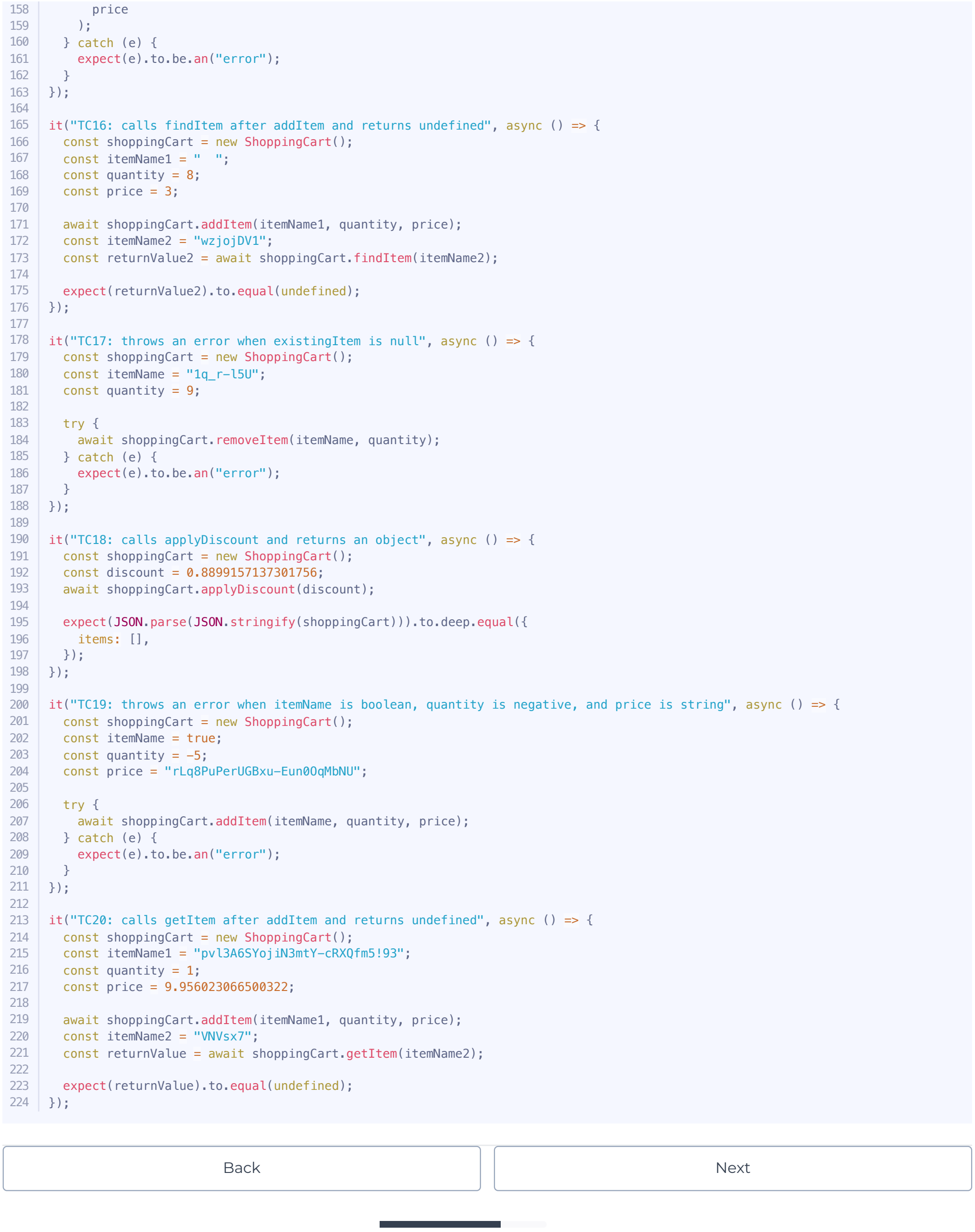
30. If you need to identify and group similar test cases together based on certain criteria or metrics, how could it be done?

Please provide a detailed explanation of your criteria or metrics of clustering. You can check all test cases below. *

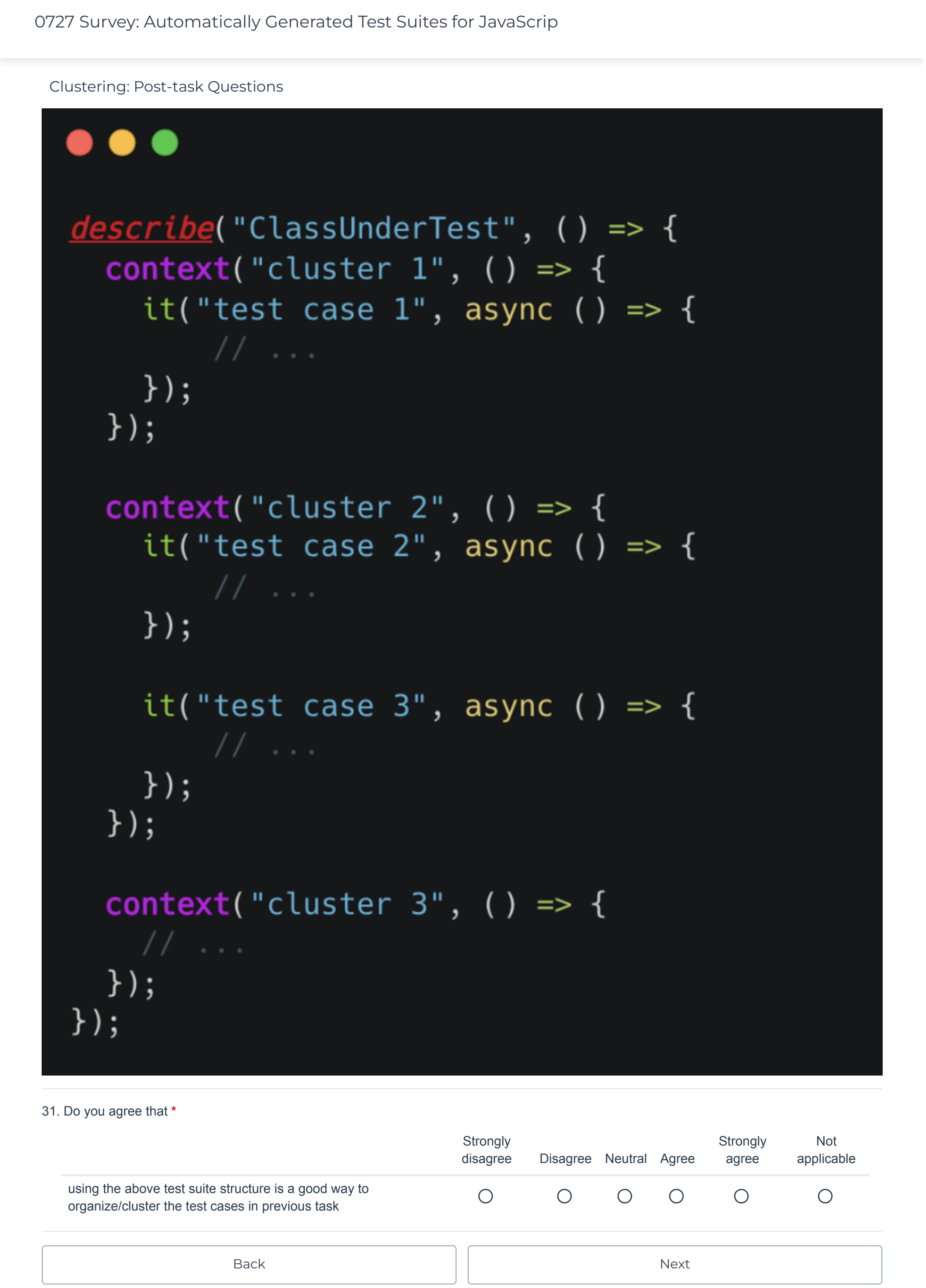
Here are all the test cases you will use in this task. Find the class under test here.

```
it("TC1: throws an error when itemName is null", async () => {
  const shoppingCart = new ShoppingCart();
  const itemName = null;
  const quantity = 6;

  try {
    await shoppingCart.removeItem(itemName, quantity);
  } catch (e) {
    expect(e).to.be.an("error");
  }
});

it("TC2: throws an error when discount is boolean", async () => {
  const shoppingCart = new ShoppingCart();
  const discount = false;

  try {
    await shoppingCart.applyDiscount(discount);
  } catch (e) {
    expect(e).to.be.an("error");
  }
});

it("TC3: throws an error when itemName is boolean and quantity is negative", async () => {
  const shoppingCart = new ShoppingCart();
  const itemName = false;
  const quantity = -4.463676586368846;

  try {
    await shoppingCart.removeItem(itemName, quantity);
  } catch (e) {
    expect(e).to.be.an("error");
  }
});

it("TC4: calls getTotalPrice and returns 0", async () => {
  const shoppingCart = new ShoppingCart();
  const returnValue = await shoppingCart.getTotalPrice();

  expect(returnValue).to.equal(0);
});

it("TC5: calls getItem and returns undefined", async () => {
  const shoppingCart = new ShoppingCart();
  const itemName = "f7TRlPDk8rN_1QhwDGbjrD0RS";
  const returnValue = await shoppingCart.getItem(itemName);

  expect(returnValue).to.equal(undefined);
});

it("TC6: throws an error when itemName is boolean", async () => {
  const shoppingCart = new ShoppingCart();
  const itemName = true;
  const quantity = 5;

  try {
    await shoppingCart.removeItem(itemName, quantity);
  } catch (e) {
    expect(e).to.be.an("error");
  }
});

it("TC7: throws an error when itemName is function", async () => {
  const shoppingCart = new ShoppingCart();
  const itemName = () => {};

```

```javascript
  try {
    const returnValue = await shoppingCart.findItem(itemName);
  } catch (e) {
    expect(e).to.be.an("error");
  }
});

it("TC8: throws an error when itemName is positve, quantity is string, and price is string", async () => {
  const shoppingCart = new ShoppingCart();
  const itemName = 9;
  const quantity = " ";
  const price = "QAvFGJhRb7V89b";

  try {
    await shoppingCart.addItem(itemName, quantity, price);
  } catch (e) {
    expect(e).to.be.an("error");
  }
});

it("TC9: calls getItems and returns empty array", async () => {
  const shoppingCart = new ShoppingCart();
  const returnValue = await shoppingCart.getItems();

  expect(JSON.parse(JSON.stringify(returnValue))).to.deep.equal([]);
});

it("TC10: throws an error when itemName is array", async () => {
  const shoppingCart = new ShoppingCart();
  const itemName = ["FLxn4T3hFmo_pdwa"];

  try {
    const returnValue = await shoppingCart.getItem(itemName);
  } catch (e) {
    expect(e).to.be.an("error");
  }
});

it("TC11: calls clearCart and return an object", async () => {
  const shoppingCart = new ShoppingCart();
  await shoppingCart.clearCart();

  expect(JSON.parse(JSON.stringify(shoppingCart))).to.deep.equal({
    items: [],
  });
});

it("TC12: throws an error when itemName is number", async () => {
  const shoppingCart = new ShoppingCart();
  const itemName = 2;
  const quantity = 1;
  const price = 3;

  try {
    await shoppingCart.addItem(itemName, quantity, price);
  } catch (e) {
    expect(e).to.be.an("error");
  }
});

it("TC13: throws an error when quantity is string and price is negative", async () => {
  const shoppingCart = new ShoppingCart();
  const itemName = "kzExxpeYXazeWf9mt1jS-lYsz_VLg";
  const quantity = "3bBWPprqh6-UQhXbeB3JDd3ZjZlxM";
  const price = -9;

  try {
    await shoppingCart.addItem(itemName, quantity, price);
  } catch (e) {
    expect(e).to.be.an("error");
  }
});

it("TC14: calls getItemCount after clearCart and returns 0", async () => {
  const shoppingCart = new ShoppingCart();
  await shoppingCart.clearCart();
  const returnValue = await shoppingCart.getItemCount();

  expect(returnValue).to.equal(0);
});

it("TC15: throws an error when price is negative", async () => {
  const shoppingCart = new ShoppingCart();
  const itemName = "eyAo";
  const quantity = 3;
  const price = -5;

  try {
    const returnValue = await shoppingCart.validateInput(
      itemName,
      quantity,
```



```
      price
    );
  } catch (e) {
    expect(e).to.be.an("error");
  }
});

it("TC16: calls findItem after addItem and returns undefined", async () => {
  const shoppingCart = new ShoppingCart();
  const itemName1 = "  ";
  const quantity = 8;
  const price = 3;

  await shoppingCart.addItem(itemName1, quantity, price);
  const itemName2 = "wzjojDV1";
  const returnValue2 = await shoppingCart.findItem(itemName2);

  expect(returnValue2).to.equal(undefined);
});

it("TC17: throws an error when existingItem is null", async () => {
  const shoppingCart = new ShoppingCart();
  const itemName = "1q_r-l5U";
  const quantity = 9;

  try {
    await shoppingCart.removeItem(itemName, quantity);
  } catch (e) {
    expect(e).to.be.an("error");
  }
});

it("TC18: calls applyDiscount and returns an object", async () => {
  const shoppingCart = new ShoppingCart();
  const discount = 0.8899157137301756;
  await shoppingCart.applyDiscount(discount);

  expect(JSON.parse(JSON.stringify(shoppingCart))).to.deep.equal({
    items: [],
  });
});

it("TC19: throws an error when itemName is boolean, quantity is negative, and price is string", async () => {
  const shoppingCart = new ShoppingCart();
  const itemName = true;
  const quantity = -5;
  const price = "rLq8PuPerUGBxu-Eun0OqMbNU";

  try {
    await shoppingCart.addItem(itemName, quantity, price);
  } catch (e) {
    expect(e).to.be.an("error");
  }
});

it("TC20: calls getItem after addItem and returns undefined", async () => {
  const shoppingCart = new ShoppingCart();
  const itemName1 = "pvl3A6SYojiN3mtY-cRXQfm5!93";
  const quantity = 1;
  const price = 9.956023066500322;

  await shoppingCart.addItem(itemName1, quantity, price);
  const itemName2 = "VNVsx7";
  const returnValue = await shoppingCart.getItem(itemName2);

  expect(returnValue).to.equal(undefined);
});
```



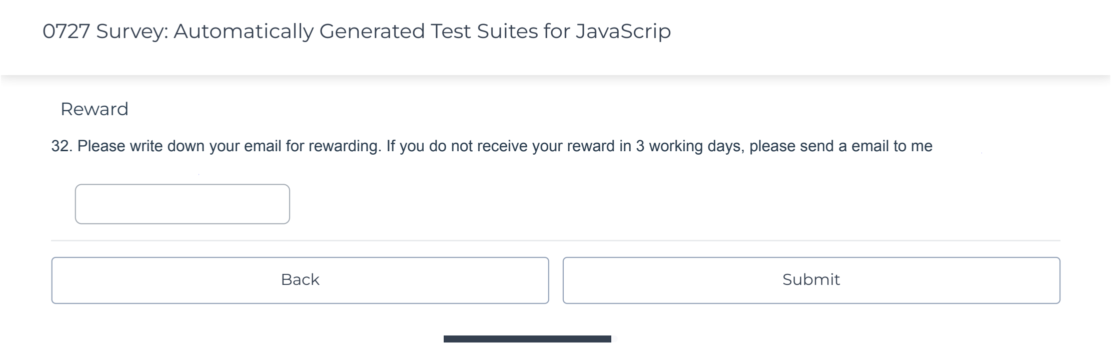
0727 Survey: Automatically Generated Test Suites for JavaScrip

Clustering: Post-task Questions

```
describe("ClassUnderTest", () => {
  context("cluster 1", () => {
    it("test case 1", async () => {
      // ...
    });
  });

  context("cluster 2", () => {
    it("test case 2", async () => {
      // ...
    });

    it("test case 3", async () => {
      // ...
    });
  });

  context("cluster 3", () => {
    // ...
  });
});
```

31. Do you agree that *

	Strongly disagree	Disagree	Neutral	Agree	Strongly agree	Not applicable
using the above test suite structure is a good way to organize/cluster the test cases in previous task	○	○	○	○	○	○



0727 Survey: Automatically Generated Test Suites for JavaScrip

Reward

32. Please write down your email for rewarding. If you do not receive your reward in 3 working days, please send a email to me