

Difference and Application of White-Box, Black-Box, and Grey-Box Testing

Vedran Marić¹

¹Faculty of Mechanical Engineering, Computing and Electrical Engineering University of Mostar, Bosnia and Herzegovina

Abstract In this paper, we explore key differences and practical applications of white-box, black-box, and grey-box testing methodologies in software development. Through theoretical comparison and real-world implementation on a test project, we demonstrate how each method serves a unique role in ensuring software quality. Testing is conducted on a web application developed with React Native and Supabase as the backend, where users can post and manage personal comments. This app serves as a case study for applying each testing approach. White-box testing was used to verify internal logic and data handling. Black-box testing evaluated user-facing features, such as comment posting. Grey-box testing combined knowledge of both to assess security and integration with the backend. This paper aims to clarify when and why each testing type should be used, particularly in small-scale projects as the one used as a test case. This serves as a practical guide for students and developers to understand and apply testing strategies more effectively in their own applications.	Article history Received: 15.09.2025. Revised: 19.12.2025. Accepted: 12.01.2026. Keywords Grey-Box, White-Box, Black-Box.
--	--

1 Introduction

For every product development, testing is a crucial step in the development life cycle. In software development, thorough testing is essential to ensure the quality, functionality, and security of applications. The most popular testing strategies, including white-box, black-box, and grey-box testing, represent foundational approaches used by developers and testers. Each one offers unique advantages and is suited for specific stages or aspects of testing [1].

1.1 Introduction to Testing

In software testing, we analyze a software item to detect differences between existing and required conditions (i.e., bugs) and to evaluate its features. Software testing should be done iteratively throughout the development process.

Software testing is one of the "verification and validation" (V&V) software practices. Verification (the first V) is the process of evaluating a system or component to determine whether the products of a

given development phase satisfy the conditions imposed at the start of that phase.

Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements [2].

Differences between white-box, black-box, and grey-box testing can be conceptualized by categorizing them into several levels of opacity [3] based on their access to or knowledge of the system's internal structure. In white-box testing, the tester has complete transparency, meaning full visibility into the source code and internal logic, enabling a detailed examination of how the application functions internally. Developers themselves mostly do this type. Black-box testing, in contrast, treats the system as entirely opaque: the tester interacts with the software solely through inputs and expected outputs, without any insight into its implementation. This makes it suitable for larger-scale projects, as tasks can be divided among other employees rather than relying solely on developers.

Contact Vedran Marić, vedran.maric@fsre.sum.ba, Faculty of Mechanical Engineering, Computing and Electrical Engineering, University of Mostar

©2026 by the Author(s). Licensee IJISE by Faculty of Mechanical Engineering, Computing and Electrical Engineering, University of Mostar. This article is an open-access and distributed under the terms and conditions of the CC BY 4.0 (<https://creativecommons.org/licenses/by/4.0/>)

Grey-box testing lies between these extremes, offering partial transparency. The tester has limited knowledge of internal components, which is often sufficient to design more informed test cases without full access to the source code. This opacity gradient affects not only the techniques used but also the types of bugs likely to be discovered.

1.2 White-Box Testing

White-box testing involves a thorough examination of an application's internal workings, typically performed by the developers themselves. The tester has full access to the source code, algorithms, control flow, data structures, and essentially everything that the application is composed of. The tester is also very well-informed about every aspect of how the software actually works, even though he was not involved in its development. This approach enables detailed testing of logical paths, branches, loops, and conditions. It is particularly useful for identifying internal errors, such as hidden bugs or unreachable code, that may not be revealed through external testing alone [3].

1.3 Black-Box Testing

Black-box testing is focused on the inputs and outputs of a software system. The tester does not need to know the internal code or architecture. Validation of the system's behavior is done by examining its response to a variety of input conditions, where the software needs to meet specified functional requirements [3]. This method is ideal for simulating user behavior and identifying issues such as incorrect outputs, missing functionalities, or interface defects. It is widely used for acceptance and system-level testing.

1.4 Grey-Box Testing

Grey-box testing is a hybrid approach between white-box and black-box testing, where the tester has partial knowledge of the internal code but no access to it. The tester is familiar with the internal structure, having access to design documents and architectural diagrams, but not the operating source code. This limited level of insight allows for more targeted testing than black-box methods, while still maintaining a user-oriented perspective, which is

hard to achieve with white-box testing. Grey-box testing is especially effective in integration testing and detecting security vulnerabilities [4].

2 Methodology Overview

The practical analysis in this paper is based on a demo web application that contains numerous issues and irregularities within different sections. The application is developed using React Native and Supabase and allows users to log in, post personal comments, retrieve them from the database, delete them, and register new users. Previous works from other researchers inspired this application design and test [5], [6]. To demonstrate the difference and practical application of various testing types, three test strategies [7] were applied to the same demo app: white-box, black-box, and grey-box testing. As there are multiple variations of each testing type, only one was selected from each method. Each test attempted to find as many irregularities as possible. The final results from this testing methodology are presented in a table in the Conclusion section, along with appropriate explanations.

White-box testing involved manual inspection of the internal logic of JavaScript functions, such as comment submission, user authentication, and database queries [7]. Console logs and manual assertions were used to verify expected outcomes. This was done using Visual Studio Code and Google Chrome. Each project file was inspected line-by-line, as requested by the line-by-line method [8].

Black-box testing is generally focused on the application's UI/UX layer. This test evaluated whether users could submit, view, and delete comments without errors, without peeking or accessing the code. This approach simulated how an end-user interacts with the app [3]. For the test, a locally hosted demo with a functional database connection was required, and the tester was to act as a regular user of the app, with no backend knowledge of the app [2].

Grey-box testing combined insights into the backend structure with external testing. It was used to assess security and data integrity, for example, by verifying whether unauthorized users could manipulate data or bypass validation [4].

More specifically, if unauthorized users could write their own or access other users' comments, for grey-box testing, a tester should receive a briefing regarding the web application's structure, database, UML, or other documents, and would be asked to try to manipulate the application in a harmful or disruptive manner [8].

All these methods were conducted iteratively to ensure comprehensive test coverage, reflect real-world testing scenarios, and facilitate comparison of results across methods.

2.1 Demo App

This section is a dedicated overview of a demo application used for white-box, grey-box, and black-box testing. The application's source code is available on GitHub; the link is provided in the Final Statements section (Figure 1).

2.1.1 Frontend

2.1.1.1 Register Page

The Register page includes a user authentication system that requires an email address and a password (Figure 2). Once the required information is provided, React sends the data to Supabase's backend, where all Accounts are linked to their respective email addresses, and there are no other obstacles to creating secondary or tertiary accounts.

```
const Stack = createNativeStackNavigator();

export default function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen
          name="Login"
          component={Login}
        />
        <Stack.Screen name="Register"
          component={Register} />
        <Stack.Screen name="Home" component={Home}
        />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

Figure 1 Demo application project main App.js. All pages are navigated through React's Stack navigator

Figure 2 Register page of the demo application

When a user enters their email address and submits the registration form, the system checks whether an account with that email address already exists in Supabase's database. If no account is found, a new user is created in the authentication database, and the individual is successfully registered.

2.1.1.2 Login Page

When attempting to log in, the application checks whether the account actually exists in Supabase's database. If a match is found and both credentials are verified, the user is granted access to the application and redirected to the Home page. This basic process is quite similar to that of most websites, making it an ideal testing ground for our various testing methods (Figure 3).

Figure 3 Login page of the demo application

2.1.1.3 Home Page

On this page, users can write down different texts without text-size limits. Users' texts are stored within the Supabase database. Once the message is sent to the backend and stored in Supabase's database, it is automatically associated with the authenticated user's unique identifier (Universally Unique Identifier or UUID), which is automatically generated for every user in Supabase. This way, we can associate each comment with its respective user. All texts are stored without any boundaries or limits for users (Figure 4).

The list of all previously submitted messages from this user is populated by retrieving all database entries associated with the user's UUID, as mentioned earlier. Clicking the bin icon in React sends an SQL DELETE request to the database for the selected message.

The third feature is a debugging tool that prints the raw JSON response from Supabase when fetching user comments. This JSON contains all text messages associated with the user's UUID and can be viewed in the browser console in the original format received from Supabase.

2.1.2 Backend

2.1.2.1 Database

The database used for this demo project is a free version of Supabase. Supabase is an open-source backend-as-a-service platform that offers a convenient, scalable alternative to traditional database solutions.

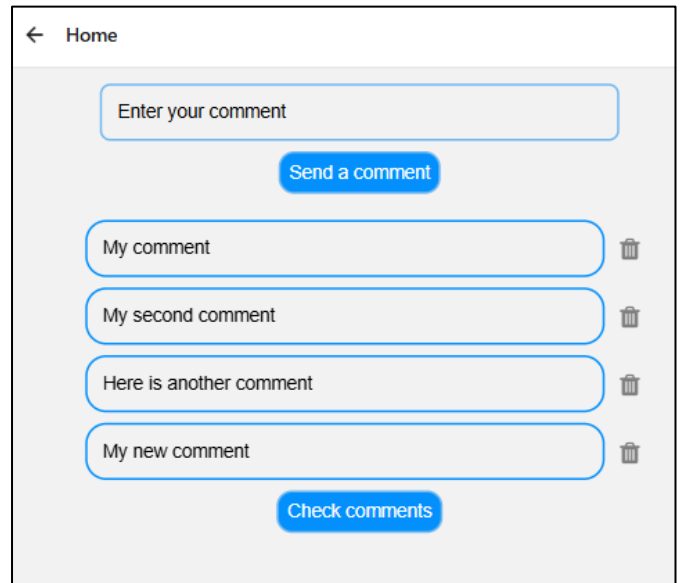


Figure 4 Main page of the demo application for posting and deleting user comments

It is built on top of PostgreSQL and offers many functionalities required by application developers, including APIs, real-time data syncing, and built-in authentication and authorization, which we use in our test application. Supabase is especially suitable for prototyping and building small-scale applications [9].

In our test application, the Row-Level Security (RLS) feature is enabled to enforce granular access control over the data [10], [11]. RLS ensures that users can only access or modify the records that belong to them. This way, we have a strong security layer crucial for maintaining user privacy and data isolation. This is particularly important when working in multi-user environments (Figure 5).

TekstKorisnika			Disable RLS	Create policy
NAME	COMMAND	APPLIED TO		
Enable delete for users based on user_id	DELETE	public		
Enable insert for authenticated users only	INSERT	authenticated		
Enable read access for all users	SELECT	public		
Update texts	UPDATE	authenticated		

Figure 5 RLS policy settings in Supabase

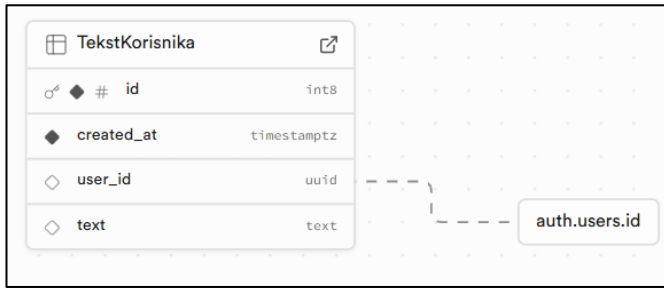


Figure 6 Database table containing user messages on Supabase

As previously mentioned, our application is using Supabase's built-in method for email and password authentication. When a user registers or logs in using this mechanism, they are authenticated and provided with a unique identifier known as a UUID (Universally Unique Identifier). This UUID is then used as a foreign key to associate individual records in the database with specific users, which, in our case, are texts they have written and decided to store in the database. The schema of our database is deliberately kept simple for demonstration purposes. It consists of a single table that stores user-submitted comments. Each comment record has a text content field and a metadata field for the UUID of the user who created it.

The overall schema of our database supports basic CRUD (Create, Read, Update, Delete) operations. This is enabling us to test our box software testing methodologies in this field as well (Figure 6).

3 Overview of Box Testing

3.1 Testing in Essence

When a software system grows in complexity, more levels of testing are necessary to ensure that the system functions correctly, reliably, and securely across its various components [8]. Table 1 provides an overview of common software testing types. We can see that the most significant differences are in the general scope, opacity, and the type of testers involved in the testing process.

This classification is essential for understanding not only what is being tested, but also how and by whom the testing is performed. We can see that our box testing is only one factor to consider when choosing the appropriate type of testing.

Table 1 Different types of software testing

Testing Type	Specification	General Scope	Opacity
Unit	Low-Level Design Actual Code Structure	Small unit of code no larger than a class	White-box
Integration	Low-Level Design High-Level Design	Multiple classes	White-box Black-box
Functional	High-Level Design	Whole product	Black-box
System	Requirements Analysis	Whole product in representative environments	Black-box
Acceptance	Requirements Analysis	Whole product in the customer's environment	Black-box
Beta	Ad hoc	Whole product in the customer's environment	Black-box
Regression	Changed Documentation High-Level Design	Any of the above	Black-box White-box

Each testing type is associated with a specific specification that defines the expected behavior of the software being tested. Exact specifications vary depending on the application's primary purpose and user requirements. This means different testing methods must be used depending on the project's scope and requirements.

Unit testing, for example, focuses on small, isolated segments of code, often no larger than a single function or class. It is typically carried out by the same developer who wrote the code. In contrast, system or acceptance types of testing examine the complete product in an environment resembling real-world usage and are often performed by independent testers or customers with no backend knowledge of the system [8].

Two critical dimensions are highlighted in this testing classification:

Opacity – As mentioned in the introduction, it refers to the degree of access the tester has to the application's internal workings. A white-box test provides full visibility into the source code and internal logic, enabling the tester to craft detailed, precise test cases [8]. A black-box test treats the system as a black box, validating outputs against

expected behavior without knowledge of the internal implementation [12]. Grey-box testing exists in between, combining limited code visibility with functional testing approaches.

Scope – It can range from granular unit-level tests to comprehensive system evaluations. As the scope increases, the focus shifts from internal logic to integration, usability, and practicality in real-world use.

As seen in Table 1, the segmentation of tests into white-box, black-box, and grey-box categories is fundamentally a categorization by level of opacity. It gives us an insight into how much a tester can observe and interact with the code's internal logic. This classification provides a useful framework for selecting the appropriate testing strategy at each stage of development, enabling us to create distinct roles within a development team. This approach enables a higher-quality testing pipeline without placing all the testing weight solely on application programmers. This is crucial for creating a comprehensive testing strategy that enables early fault detection, potentially increasing our reliability in real-world conditions.

In essence, the segmentation of tests into boxes (white-box, black-box, and grey-box) is a form of test segmentation based on the level of opacity. Higher opacity provides the tester with more insight into the application's actual code, while lower opacity yields a more vague description, if any.

3.2 White-Box Testing

White-box testing is also known as structural testing, clear-box testing, and glass-box testing. White-box testing involves examining the code's internal structure. When the internal structure of a product is known, tests can be conducted to ensure that the internal operations are performed according to the specification and all internal components have been adequately exercised [7].

Types of White-box Testing [7]:

- Code coverage
- Segment coverage
- Branch Coverage or Node Testing

- Compound Condition Coverage
- Basis Path Testing
- Data Flow Testing (DFT)
- Path Testing
- Loop Testing

Manual code coverage was used in our test example.

The following are the problems with white-box testing [7]:

- Potentially most exhaustive of the three, because it is not possible to test each and every path of the loops in the program.
- Since test cases are written on the code, specifications missed out in coding would not be revealed.
- Because a skilled tester is needed to perform white-box testing, the costs are increased.
- Sometimes it is impossible to look into every nook and corner to find out hidden errors that may create problems, as many paths will go untested.
- It is difficult to maintain white-box testing as the use of specialized tools like code analyzers and debugging tools are often required.
- Requires internals to be fully known.

In the language of V&V, white-box testing is often used for verification (i.e., are we building the software correctly?) [2].

3.3 Black-Box Testing

Black-box testing is also known as functional testing. It treats the system as a "Black-box," so it doesn't explicitly use knowledge of the internal structure or code. The code and test engineer need not know the internal working of the "Black-box" or application. Here, the primary focus is on the system's overall functionality.

Types of Black-box Testing [7]:

- Graph-Based Testing Methods
- Error Guessing
- Boundary Value Analysis
- Equivalence Partitioning
- Behavioral Testing

- Random Testing/Stochastic Testing
- Syntax Testing
- Stress Testing

The following are the problems with black-box testing:

- Low granularity
- It can be tested only by trial and error
- The test inputs need to be from a large sample space.
- Blind Coverage - It is difficult to identify all possible inputs in a limited testing time. So, writing test cases is slow and difficult
- Chances of having unidentified paths during this testing
- The test can be redundant if the software designer has already run a test case.
- The test cases are difficult to design.

In the language of V&V, black-box testing is often used for validation (are we building the right software?) [2].

3.4 Grey-Box Testing

The grey-box methodology is a ten-step process for testing computer software. The methodology starts by identifying all the input and output requirements of a computer system. This information is documented in the software requirements. The grey-box methodology uses automated software testing tools to generate unique software tests. The toolset generates module drivers and stubs, relieving the software test engineer from the need to create this code manually.

Grey-box testing uses assertion methods to preset all the conditions required before testing a program. Testing with formal specification languages is a widely used technique for ensuring that a core program is highly correct. If the requirement specification language is used to specify the requirement, it would be easy to execute the stated requirement and verify its correctness. Grey-box testing will use the predicates and verifications defined in the requirement specification language as inputs to the requirements-based test case generation phase [7].

Different forms of grey-box testing techniques are frequently used:

- Matrix Testing
- Strategy for Gray box Regression testing
- Pattern Testing

3.4.1 Matrix Testing

In matrix testing, the project status report is provided. The idea of beginning your testing activities with a list of variables used in the software is not new. Basically, the CRUD (create, read, update, and delete) method. It starts with developers defining all the variables that exist in their programs. Each variable will have both inherent technical and business risks and can be used at varying frequencies throughout its life cycle.

3.4.2 Software Regression Testing

Regression testing: testing performed after making a functional improvement or repair to the program. Its purpose is to determine if the change has regressed other aspects of the program. This can be accomplished by executing the following testing strategies:

- Retest all: Rerun all existing test cases
- Retest Risky Use Cases: Choose baseline tests to rerun by risk
- Retest By Profile: Choose baseline tests to rerun by allocating time in proportion to operational profile
- Retest Changed Segment: Choose baseline tests to rerun by comparing code changes. (White-box strategy)
- Retest within Firewall: Choose baseline tests to rerun by analyzing dependencies. (White-box strategy)

3.4.3 Pattern Testing

Pattern testing is best accomplished by analyzing historical data on previous system defects. The analysis template will include specific reasons for the defect, necessitating a thorough analysis of the system. Unlike black-box testing, which tracks failure types, grey-box testing delves into the code to determine why a failure occurred. This information is extremely valuable, as future test case design will be proactive in identifying other failures before they

occur in production. The coding structure in place influences grey-box test case design [7].

4 Testing Conduct

4.1 White-Box Testing

During white-box testing, every file was inspected internally [1] for logical errors and security flaws. Although the website was completely functional, its examination revealed functions that, although written, were left unused and served no purpose in the application.

In addition to logical flaws, the code shows inconsistencies in its writing, as no standard programming practices were followed.

Most notably, database connections are not isolated into separate files for use across multiple pages; instead, they are embedded on each page. This is contrary to regular programming practices, making the code more confusing, increasing the risk of a security breach, and forcing us to write almost identical code multiple times on different pages. The entire code review was performed line-by-line as advised [1].

Conducting white-box testing on our demo allowed us to identify these errors internally and implement improvements before launching to the production or publishing stages (Figure 7).

4.2 Black-Box Testing

Using this testing method, we have no access to the source code and can only attempt to manipulate the application in ways that future users can as well, making this part more UI/UX-oriented than other

```

async function fetchComments() {
  const { data: userData, error: userError } =
await supabase.auth.getUser();
  const { data, error } = await supabase
    .from("TekstKorisnika")
    .select("*")
    .eq("user_id", userData.user.id);
  if (error) {
    console.log(error);
  } else {
    console.log(data);
    setComments(data);
  }
}

```

Figure 7 Database connections are not isolated but incorporated into every page where contact with Supabase is required

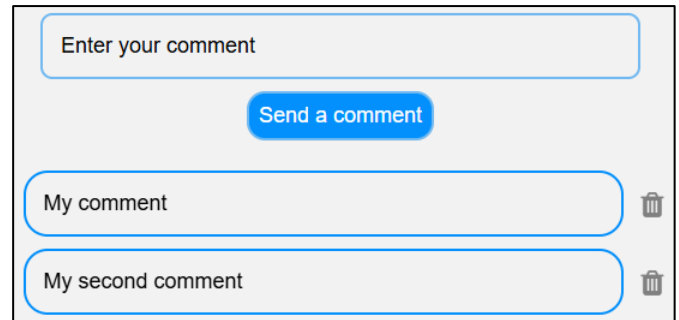


Figure 8 Create, Read, and Delete functionalities for messages are missing. Update the ability to fulfill CRUD requirements

testing methods. This method is also most commonly used to attack the application as an outside user [1].

For our demo application, we prepared no documentation to help us understand its working logic. The test is conducted the same way a first-time user would use the application (Figure 8).

This way, with black-box testing, we confirmed that our users can successfully log in, register on our application, and write and delete comments. This was our expected and received output [2].

We also noticed a few shortcomings and pointless functionalities. Although all implemented functions work as intended (from a user perspective), our demo application has failed to comply with the basic CRUD system. There is simply no logic implemented for editing profile information or messages that the user has previously written. Such inputs are a necessity in modern applications, and their absence is easily detectable by the user, thus making them detectable in our black-box testing as an extension.

Additionally, we have an extra feature that provides no value to our project but instead confuses users, namely, checking users' messages in the browser's console. This is an unnecessary function, as messages are already displayed on the application's page in a much more readable format. At the same time, most users would not even be able to find these messages through console logs (Figure 9).

```

function Logout() { //logout
  navigation.navigate('Login');
}

```

Figure 9 Unused logout function

```

0:
  created_at: „2025-05-24T11:00:37.600002+00:00“
  id: 12
  text: „My comment“
  user_id: „d04a445a-c3ff-4ab9-93a6-ed480fc66b31“
  > [[Prototype]]: Object
1:
  created_at: „2025-05-24T11:00:42.720002+00:00“
  id: 13
  text: „My second comment“
  user_id: „d04a445a-c3ff-4ab9-93a6-ed480fc66b31“
  > [[Prototype]]: Object
    
```

Figure 10 Messages shown through the web browser's console

This method of displaying information is practical during application development, but becomes pointless and unwanted in the final versions of the product (Figure 10).

Additionally, no error messages are displayed when the user enters an empty message, increasing the user's uncertainty about whether the comment has been received and saved. During this testing method, we attempted an SQL injection as instructed by Georgia Weidman [12]. SQL injection covers 15% of the most common application vulnerabilities [13]. These attempts, however, were unsuccessful in breaking the application. Although unsuccessful, had our application really contained such an oversight, it would have been detected by black-box testing.

Black-box testing mostly allowed us to identify errors that would be the first inconveniences experienced by users of the application and would directly impact users' quality of experience. Security and other concerns are of secondary importance and only partially testable.

4.3 Grey-Box Testing

For grey-box testing, the first requirement is to have pre-existing knowledge of the backend schema (the Supabase table 'TekstKorisnika') and of the relevant logic (or UML documentation). This is done so we can validate that the correct data (in our case, user

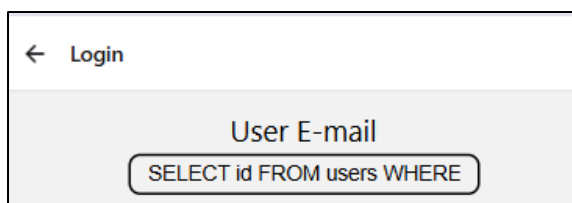


Figure 11 SQL injection attempt

comments) is being queried and displayed in the proper component (FlatList), where messages are displayed by design.

Knowing our database security settings, we know that for registering new users, no email confirmation is required, thus allowing us to register using emails that are not our own. This is a serious security concern that needs to be addressed in the final product. Although it is possible to reveal such issues through both black-box and white-box testing, grey-box testing is particularly effective at identifying them much more quickly (because we are familiar with this security flaw from knowing the database settings), saving us valuable time and other resources (Figure 11).

Grey-box testing also helped us reveal that deleted comments remained in the cache and needed to be refetched for them to actually appear deleted to the user. This is an inconvenience that would be rather difficult to detect using white-box testing; however, black-box testing would most likely identify the stated oversight more quickly.

5 Results

A comprehensive summary of testing outcomes from various research methods and our experimental testing can be consolidated into Table 2. The table below outlines the potential for detecting specific issues using black-box, white-box, and grey-box testing strategies. Each cell indicates whether the method can identify the given issue.

Table 2 Possibility for each issue to be detected

Method / Issues	Black-box	White-box	Grey-box
CRUD operations	Partial	Yes	Yes
Unused functions	No	Yes	No
Fake user spam	Yes	Yes	Yes
Unisolated database connections	No	Yes	No
UI problems	Yes	Yes	Yes
SQL injection	Yes	Yes	Yes

The table illustrates the unique strengths and limitations of each testing method in the cases we examined. Some vulnerabilities and inefficiencies are detectable with nearly any approach (such as issues related to CRUD operations and SQL injection). In contrast, others, such as unused functions in the source code or unisolated backend logic, remain completely invisible unless the tester has direct access to the code, which can only be achieved through white-box testing.

Black-box testing was unable to partially check all CRUD operations, as from a user-oriented perspective, it is not possible to determine whether deleting a comment on this website actually deletes it from the database or only flags it as "hidden" for the end user.

Unused functions in this demo application cannot be found through black-box and grey-box testing, as we do not have access to the application's source code, which makes them easy to investigate in Visual Studio Code and similar software. Similar situation is with unisolated database connections with the important exception that unisolated connection CAN be found through sending numerous random database requests through the console, but in this case all requests are sent blindly, and it is not possible to conduct a serious testing where all connections are checked without having greater insight into the code itself, which is not provided in these testing scenarios.

5.1.1 CRUD Operations

5.1.1.1 Black-Box Testing

Black-box testing involves testing the Create, Read, Update, and Delete functionalities by directly interacting with the application interface to identify inconsistencies. After a successful login (or registration), the user can write new comments, which, after being sent to the database, are displayed immediately and each time the user logs in to their account. This confirms that CRUD operations, Create and Read, are indeed functional and testable through black-box testing.

Clicking the trash icon next to the message displayed removes it from the screen. Once the user is logged in again, the removed message is no longer available. This confirms that we can test the Delete operation as

well, although only from a user's perspective. The user's perspective is a limiting factor: the message is deleted for the user, but it is not possible to confirm whether it is deleted in the database or only flagged as removed. The update operation is not available in our demo application because messages and account information provided immediately cannot be altered. Inability to change any information as a user confirms that we can detect malfunctions with the Update operation through black-box testing.

5.1.1.2 White-Box Testing

White-box gives us full insight into the background code behind our application. To detect issues with CRUD operations, basic programming knowledge is required, making this method more complex than the black-box alternative (Figure 12).

With full insight into the app, we can check for functions responsible for implementing CRUD operations. Through this method, it is possible to verify the existence of the database in which all data is initially stored. Database existence can be verified by checking the Supabase account, and the proper connection is verified in the supabaseClient.js file. Functions required for the Create operation can be found within separate page files, such as the one for submitting a new comment within the Home.js file. All this confirms that the CRUD operation Create can be verified through white-box testing.

Inability to find any functions or files dedicated to the Update operation, where they are expected to be them is a sign that should alarm every experienced programmer. This confirms that the Update function can also be verified via this method. Delete and Read functions are tested in a similar way.

```

    async function sendComment(){ //sends new
comment into the database
(...)
    const { error } = await supabase
      .from("TekstKorisnika")
      .insert([
        {
          text: newComment,
          user_id: userData.user.id,
        }
      ]);
  (...)
```

Figure 12 Create a new comment function SQL segment

```

    async function fetchComments() { //fetching
    comments
    const { data: userData, error: userError } =
    await supabase.auth.getUser();

    if (userError) {
        console.log("Error with the user:",
    userError.message);
        return;
    }

    const { data, error } = await supabase
        .from("TekstKorisnika")
        .select("*")
        .eq("user_id", userData.user.id);

    if (error) {
        console.log(error);
    } else {
        console.log(data);
        setComments(data);
    }
    }
}

```

Figure 13 Function for "Reading" comments from the database

Looking through the project code, we can find functions dedicated to fetching and deleting comments from the database inside the Home.js file, confirming the existence of our Read and Delete methods (Figure 13).

Testing their functionality can be verified by examining the code itself, but the most common and easiest way is to start the program through the console and manually test its effectiveness. This confirms that we can test all CRUD operations through white-box testing.

5.1.1.3 Grey-Box Testing

Grey-box testing can verify all CRUD operations, as with black-box testing. The main difference is that, through using developer tools, we can check. The

```

    async function deleteComment(id) { //deleting a
    comment
    const { error } = await supabase
        .from("TekstKorisnika")
        .delete()
        .eq("id", id);

    if (error) {
        console.log("Error while deleting
    occured:", error.message);
    } else {
        console.log("Comment erased
    successfully.");
        fetchComments()
    }
    }
}

```

Figure 14 Function for deleting a comment from the database

developer tool can be opened in Google Chrome by pressing F12. In case an error occurs, we can examine the Console panel to identify the issues that have arisen. In the demo application provided, such errors can occur in the Supabase project if the database is not running. Otherwise, successful CRUD operations can be observed through the information they provide to the database. This confirms that CRUD operations can be tested through grey-box testing (Figure 14).

5.1.2 Unused Functions

5.1.2.1 Black-Box Testing

As this method provides no insight into the project's code, and the nature of unused functions is such that they are not used within the project, it is not possible to detect them via black-box testing. This can be assumed when certain functionalities are not functional from a user's perspective (such as the Edit comment option), but there is no way to know whether the edit comment function is not used or not even written in the first place.

5.1.2.2 White-Box Testing

White-box testing provides a straightforward approach for verifying unused functions. In certain source-code editors, such as Visual Studio Code, functions that are written but not used anywhere in the project are darkened in Dark mode or lightened in Light theme, making them easily noticeable to developers and testers.

Figure 15 shows the unused function Logout, which, if used anywhere in the file, would have its name in the same colour as the navigate function in the example "navigation.navigate('Login').".

This confirms that unused functions can easily be detected using white-box testing.

5.1.2.3 Grey-Box Testing

Grey-box testing suffers from the same issues as the black-box method. It is not possible to check for unused parts of the code when no insight into the

```

function Logout() { //logout
    navigation.navigate('Login');
}

```

Figure 15 Unused function Logout would be of different color than other function in code editor

code is provided. It is possible to suspect certain aspects or functions are unused, but this cannot be distinguished from not being written in the first place. As the user is unable to log out, it is possible to suspect that the Logout function is not being used; however, there is no way to know whether developers have even written the Logout function in the source code.

5.1.3 Fake User Spam

5.1.3.1 Black-Box Testing

Black-box testing is by far the easiest method to test an application's ability to check for fake spamming of fake accounts on the website. By entering the registration screen, users can provide any email address and any password with at least 6 characters, allowing them to access the website. Simply pressing the back-arrow button returns the user to the Register page, where they can register a new account using a different email. This method for creating a new account can be used indefinitely. This way, it is possible to confirm fake user spam through black-box testing.

5.1.3.2 White-Box Testing

Checking for fake user spam through the white-box method can be done in multiple ways. A more comprehensive search would identify methods and functions in the source code that prevent this issue. An easier, but less reliable, method is to open the Supabase database of all registered accounts and manually search for suspicious accounts, for example, by examining the exact date and time they were created. If hundreds of accounts are registered within minutes in a small web application such as this demo application, it is an alarm for developers and administrators to check for fake accounts. Therefore, it is possible to detect fake accounts in multiple ways using a white-box approach.

5.1.3.3 Grey-Box Testing

Testing for fake account spamming through grey-box testing is again quite similar to black-box testing, with a small exception: it provides more insight by using developer tools in an internet browser. Using the console provided in developer tools, it is possible to directly check what is sent and received from the database, confirming that a new (fake) account is

successfully created. Therefore, this vulnerability can be checked using grey-box testing.

5.1.4 Unisolated database connections

5.1.4.1 Black-box testing

An unisolated connection to the database cannot be identified through black-box testing, as it is not possible to determine which source file establishes the connection. It is, however, possible to infer an isolated connection from the website's slow loading time. This cannot be a proof in any way, as numerous other factors can cause slow loading times, and this is not always a recurring symptom; therefore, it is concluded that black-box testing cannot be used to verify the absence of unisolated database connections.

5.1.4.2 White-Box Testing

White-box is the only method able to check for isolated or unisolated database connections. Looking at Home.js and other pages using data from the database, it is clear that they all use the same connection defined in supabaseClient.js. Also, checking Supabase settings such as RLS rules and location of API keys, which is in this demo application found in the supabaseClient.js file instead of the .env file, which is a development standard [3]. Therefore, it is possible to check for unisolated database connections through white-box testing.

5.1.4.3 Grey-Box Testing

Grey-box testing can check network responses using developer tools and website loading times, but, as with black-box testing, it is not possible to determine whether the database connection is isolated.

5.1.5 UI Problems

5.1.5.1 Black-Box Testing

User interface problems are easily tested from a user's perspective using black-box testing. In the event of any UI discrepancy, the error will be noticeable on the screen. Wrong button colours and similar issues are immediately noticeable. Difficulty with this method arises when the user is unaware of all the options available on the website. Therefore, when in the case of this demo application, the edit comment option is not available, through the black-box method, it is not possible to differentiate if the UI

error is in place (such as the Edit message button being hidden or missing) or the Edit message simply is not planned or even developed. Therefore, it is possible to detect UI errors using a black-box method, although some "false positives" will also be detected.

5.1.5.2 White-Box Testing

White-box testing is beneficial for certain UI mistakes, such as two buttons with different colours that should be the same, which can be checked by inspecting the hex colour assigned to their CSS styling. However, most UI mistakes are difficult to detect from the source code. The best method is to start the program and conduct a visual inspection, which is similar to black-box testing. The only difference is that, with access to the source code, it is possible to remove any "false positives" that would otherwise occur.

5.1.5.3 Grey-Box Testing

Through grey-box testing, it is possible to verify the colors and positions of buttons and other UI elements, just as in black-box testing. However, this method has expanded capabilities to detect UI errors hidden from the user by checking the browser's developer tools console. In case all elements written in the source code are rendered correctly, the website's console would appear empty or at least without any UI rendering errors, as was the case for the demo application.

This also helps remove "false positives" that arise from a clear black-box method, as all errors are displayed in the console and can be distinguished from UI problems if the console indicates the issue is something else.

5.1.6 SQL Injections

5.1.6.1 Black-Box Testing

SQL injection can be tested through black-box testing. On the Register page, it is possible to write certain SQL codes, such as "' OR 1=1 --", which is a malicious code that could allow a user to access any account with no password if the database is not properly connected. For the demo application, this code is inserted into all input fields on the Register and Login pages, but there was no breach in the website. This way, it was demonstrated that the demo application is resistant to SQL injection, and the test was conducted using the black-box method.

5.1.6.2 White-Box Testing

White-box testing enables us to inspect Supabase settings to verify RLS settings. RLS settings provide insight into who and what can do once a request is sent to the database. Also, on the frontend, you can check the code in Login.js and Register.js for input validation. Also, dummy data can be used to simulate malicious user inputs and detect possible breaches. Therefore, white-box not only detects SQL injections but also provides great insight into which specific part is most vulnerable (Figure 16).

5.1.6.3 Grey-Box Testing

Testing for SQL injections is similar to black-box testing. However, more insight can be obtained through the developer's console. Unlike black-box testing, once a malicious code is placed (such as ' OR 1=1---) console will show a return message from the database. In the demo application, the inserted input is shown as invalid. Inserting these and similar codes, and then searching for database results, can be used

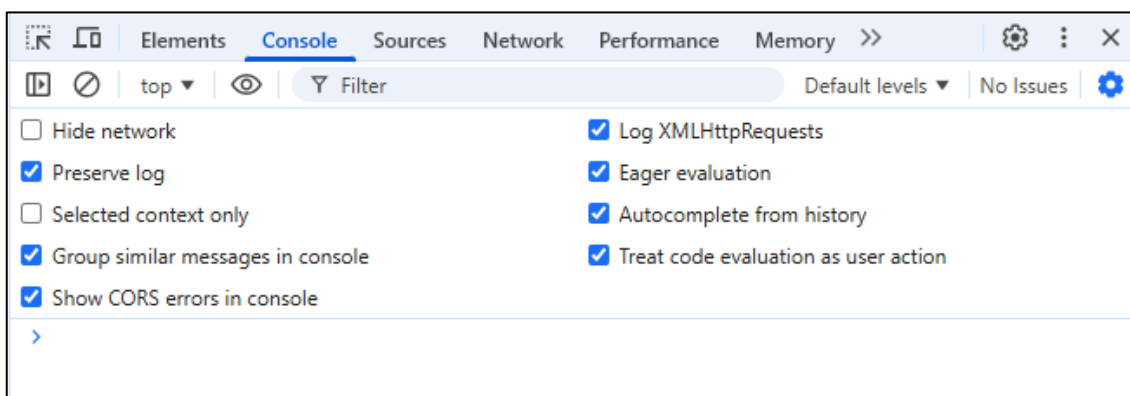


Figure 16 Clear console output for Home.js

to test for SQL injection vulnerabilities via grey-box testing.

5.1.6.4 Test Limitations

The evaluation presented in this paper is based on a single demo application developed with React and Supabase, which, although designed to be as general as possible, limits the results' generalizability. All testing activities were performed manually, without any automated testing frameworks. Furthermore, no quantitative metrics such as execution time or the number of defects detected by each testing approach were collected or analyzed.

6 Conclusion

Based on the collected data and the comparative evaluation of black-box, white-box, and grey-box testing methods, it becomes evident that each approach offers unique strengths and limitations, as summarized in Table 3. The comparison in Table 3 illustrates how each method is used to achieve different project goals under varying circumstances. Black-box testing is the quickest, especially when verifying end-user functionality and usability. White-box testing is the most demanding in terms of time and expertise, but it thus provides the most in-depth insight into the application's inner logic and code-level integrity. It is also the most difficult one to conduct on a large scale. Grey-box testing is a hybrid approach that strikes a balance between the two extremes presented by black-box and white-box methods.

Table 3 – Box testing characteristics drawn from the testing demo application

Method / Characteristic	Black-box	White-box	Grey-box
Complexity	Least complex	Most complex	Balanced
Time consumption	Smallest	Extensive	Balanced
Detection scope	User-oriented	Varies	Varies
Prior knowledge required	None	Extensive	Moderate
Unique testing method	No	No	No

Overall, it is strongly recommended to adopt multiple testing strategies. Black-box, white-box, and grey-box testing must be used together to ensure the highest software quality. Together, they offer a thorough and multidimensional assessment of the software: white-box addresses technical issues, black-box focuses on user quality of experience, and grey-box serves as a go-to alternative when certain parts are simply too large or complex to handle in detail.

While it may be technically feasible to rely on a single method (white-box testing), in practice, that is possible only for very small or micro-scale applications. The complexity of medium- and large-scale projects requires the combined use of at least two, if not all, box testing types to minimize risks and detect a broader range of issues, enabling developers to ensure overall software reliability.

That said, the choice of testing methods should align not only with the project's complexity but also with the development timeline, available resources, intended use cases, and other significant factors. Best practices, as recommended by this study and other industry literature, suggest that integrating all three testing types is the most effective approach to delivering high-quality software.

Software development has been around for many decades, and practice has shown that all three methods have their practical uses. Therefore, if possible, let's use them!

Conflicts of Interest: The author reports there are no competing interests to declare;

The data presented in this study are available on request from the corresponding author;

Source code of the demo application can be found here: <https://github.com/vedran-maric/PenTest.git>

7 References

- [1] D. Stuttard and M. Pinto, *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Indianapolis, in: Wiley, 2011.
- [2] L. Williams, "Testing Overview and Black-Box Testing Techniques," in *A (Partial) Introduction to Software Engineering Practices and Methods*, in Laurie Williams. , 2006. [Online]. Available: <https://sdc.csc.ncsu.edu/files/resources/williams-software-engineering-2011.pdf>
- [3] I. Sommerville, *Software Engineering*, Boston: Pearson Education, 2016.
- [4] E. Elkind, B. Genest, D. Peled, and H. Qu, "Grey-Box Checking," in *Formal Techniques for Networked and Distributed Systems - FORTE 2006*, E. Najm, J.-F. Pradat-Peyre, and V. V. Donzeau-Gouge, Eds., Berlin, Heidelberg: Springer, 2006, pp. 420–435. doi: 10.1007/11888116_30.
- [5] K. P. M, S. K, R. M, and K. R, "CRUD Application Using ReactJS Hooks," *EAI Endorsed Transactions on Internet of Things*, vol. 10, Mar. 2024, doi: 10.4108/eetiot.5298.
- [6] M. E. Khan and F. Khan, "A Comparative Study of White Box, Black Box and Grey Box Testing Techniques," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 3, no. 6, Jul. 2012, doi: 10.14569/IJACSA.2012.030603.
- [7] R. Saxena and M. Singh, "Gray Box Testing: Proactive methodology for the future design of test cases to reduce overall system cost," *Journal of Basic and Applied Engineering Research*, Oct. 2014. [Online]. Available: https://krishisanskriti.org/vol_image/12Jun201506060717.pdf
- [8] L. Petrović, "Analysis of OAuth 2.0 Vulnerabilities Arising from Weak Implementation Choices," *International Journal of Innovative Solutions in Engineering*, vol. 2, no. 1, p. 15, Nov. 2025, doi: 10.47960/3029-3200.2026.2.1.15.
- [9] A. Streza, "A Supa-Introduction to Supabase," Medium. Accessed: Jan. 13, 2026. [Online]. Available: <https://medium.com/@alex.streza/a-supabase-introduction-to-supabase-e551ea6708e>
- [10] C. Dar, M. Hershcovitch, and A. Morrison, "RLS Side Channels: Investigating Leakage of Row-Level Security Protected Data Through Query Execution Time," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 1–25, May 2023, doi: 10.1145/3588943.
- [11] F. Kincheloe, *Power BI Row Level Security for University Data*, Denton: TAIR Annual Conference, 2022.
- [12] G. Weidman, *Penetration testing - A Hands-on Introduction to Hacking*, San Francisco: No starch press, 2014.
- [13] D. Musa and I. Markić, "Achieving Successful Software Penetration Testing," *International Journal of Innovative Solutions in Engineering*, vol. 1, no. 1, pp. 1–9, Jan. 2025, doi: 10.47960/3029-3200.2025.1.1.1.
- [14] D. M. S. Varalakshmi, *Secure Multilevel System (FLS & RLS) for a Cyber-Physical System in Association with Data Mining*, *International Journal of Interdisciplinary Research and Innovations*, 2019.
- [15] "Row-Level Security 101: The Basics of Row-Level Security," Satori. Accessed: Jan. 13, 2026. [Online]. Available: <https://satoricyber.com/row-level-security/row-level-security-101/>
- [16] P. S. B. Bele, A. A. Harinkhede and V. M. Bhatti, Software Testing Using White-box, *International Research Journal of Innovations in Engineering and Technology (IRJIET)*, 2022.
- [17] H. Hi, "Bridge between Black Box and White Box – Gray Box Testing Technique", Accessed: Jan. 13, 2026. [Online]. Available: https://www.academia.edu/32153458/Bridge_between_Black_Box_and_White_Box_Gray_Box_Testing_Technique