International Institute of Information Technology, Bangalore

Software Testing (CS-731)
Major Project Report
Mutation Testing on Anime Review System

Submitted By:	Roll Number:
Pranjal Gawande	MT2023192
Chittaranjan Chandwani	MT2023193

<u>INDEX</u>

Project Objective

The objective of this project is to evaluate and enhance the robustness and reliability of the Anime Review System, a Java Spring Boot application, by employing mutation testing using PIT (Pitest) for Java. Mutation testing is a fault-based software testing technique designed to assess the effectiveness of existing unit test cases in identifying potential flaws and ensuring code quality.

Through this project, we aim to introduce subtle changes, or "mutants," to the codebase and determine the capability of the test suite to detect these deviations. The focus will be on analyzing the mutation score, identifying weak points in the test cases, and improving overall test coverage. This process will highlight potential gaps in the testing framework and ensure that the application adheres to high standards of correctness and reliability.

The Anime Review System, with functionalities for user and admin roles, including review management, watchlist features, and user interactions, provides an ideal candidate for applying mutation testing due to its multi-service architecture. By leveraging PIT's capabilities, this project seeks to bridge the gap between theoretical testing methodologies and practical software engineering, delivering insights that enhance both the quality of the application and the developer's expertise in advanced testing techniques.

What is Software Testing?

Software testing is the process of evaluating a software application or system to identify any defects, errors, or gaps in its functionality compared to the specified requirements. It involves executing a program or system with the intent of finding defects and ensuring that the software performs as expected under various conditions.

Software testing can be broadly categorized into two types:

1. **Manual Testing**: Performed by human testers, this involves manually executing test cases without the use of automation tools.
2. **Automated Testing**: Involves using software tools to execute test scripts and validate the software's performance.

Testing can occur at various levels of the software development lifecycle, such as unit testing, integration testing, system testing, and acceptance testing. Each level has its specific purpose and scope to ensure software quality.

Why is Software Testing required?

Software testing is crucial for several reasons:

1. **Ensures Functionality**: It verifies that the software functions as intended, meeting user and business requirements.
2. **Improves Quality**: Testing identifies and fixes bugs, ensuring a high-quality product.
3. **Enhances User Experience**: A defect-free software application provides a seamless and satisfactory user experience.
4. **Builds Confidence**: It provides stakeholders with confidence in the software's reliability and performance.

5. **Reduces Costs**: Identifying and fixing defects early in the development process is significantly less expensive than addressing issues post-deployment.
6. **Compliance with Standards**: It ensures that the software adheres to industry and regulatory standards.

Software testing is an essential activity in software development that safeguards the software's quality, reliability, and usability, thereby ensuring its success in real-world applications.

Mutation Testing

Mutation testing is a powerful fault-based testing technique that assesses the quality of a test suite by injecting small, deliberate faults, called **mutants**, into the source code. A **mutant** is a modified version of the program where specific changes are made to the code using predefined rules, known as **mutation operators**. These mutants simulate potential bugs that might occur during real-world software development.

The primary goal of mutation testing is to check if the test cases can detect and "kill" these mutants by failing when the program behaves differently from the original. If a mutant survives (i.e., the test cases do not fail), it indicates weaknesses in the test suite and the need for improvements.

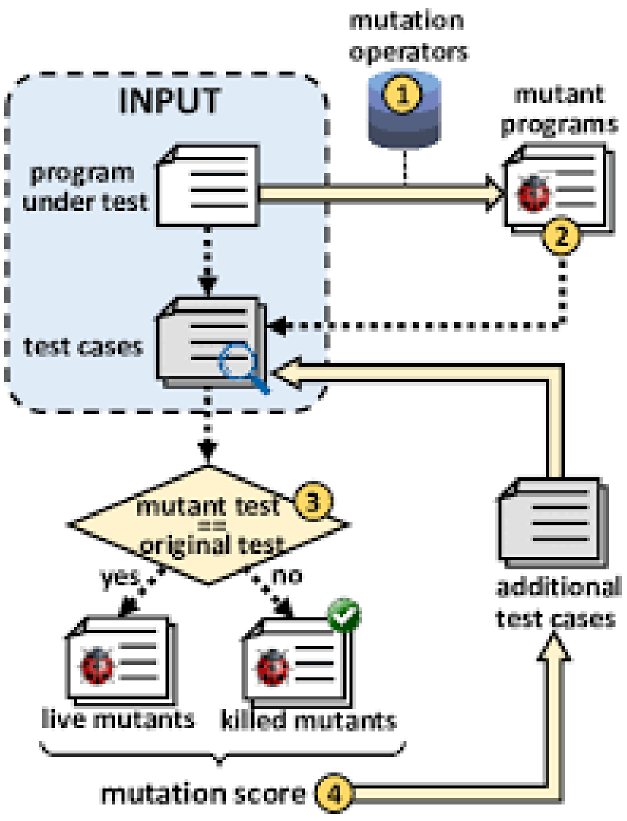


Figure 1: Mutation Testing Workflow

Strong and Weak Killing of Mutants

1. **Strong Killing of Mutants**:
   A mutant is strongly killed when a test case can distinguish the behavior of the mutant from the original program, causing a failure in the test case. Strong killing requires that the mutant's effect propagates to the program's output, which the test suite must then catch.
2. **Weak Killing of Mutants**:
   A mutant is weakly killed when the mutation affects the internal state of the program but does not necessarily lead to observable differences in the output during testing. While weak killing is an intermediate step, it signals that the mutation has some impact, though not enough to cause test failures.

Why Mutation Testing is Needed?

1. **Evaluating Test Case Robustness**: It ensures the test cases are effective in detecting subtle defects.
2. **Enhancing Code Quality**: By exposing weak test cases, it guides developers to write better, more comprehensive tests.
3. **Validating Fault Detection Capability**: Simulating real-world bugs helps verify the test suite's ability to detect various defect types.
4. **Encouraging Better Test Coverage**: It identifies untested or weakly tested parts of the code.

Types of Mutation Operators

Mutation operators define the rules for creating mutants. Common mutation operators include:

1. **Arithmetic Operator Replacement**: Alters arithmetic operators (e.g., replacing `+` with `-`).
2. **Logical Operator Replacement**: Modifies logical operators (e.g., replacing `&&` with `||`).

3. **Relational Operator Replacement**: Changes relational operators (e.g., replacing `==` with `!=`).
4. **Constant Replacement**: Replaces a constant with another value (e.g., `5` with `10`).
5. **Variable Replacement**: Substitutes one variable with another in the same scope.
6. **Conditional Operator Replacement**: Changes conditional statements (e.g., replacing `if (x > 0)` with `if (x >= 0)`).

Mutation Score

The effectiveness of mutation testing is measured using the **mutation score**, calculated as:

$$\text{Mutation Score} = \left(\frac{\text{Number of Mutants Killed}}{\text{Total Number of Mutants}}\right) \times 100$$

- A **high mutation score** indicates a robust test suite capable of detecting most faults.
- A **low mutation score** highlights weaknesses, suggesting the need to add or improve test cases.

Unit and Integration Levels of Mutation Testing

1. **Unit Testing Level**:
   - Focuses on testing individual methods or classes.
   - Verifies the correctness of small, isolated components.
   - Example: Testing the `addReview()` method to ensure it correctly adds user reviews in the Anime Review System.

2. **Integration Testing Level**:
    - Tests interactions between multiple modules or services.
    - Ensures seamless communication and data exchange across components.
    - Example: Testing the interaction between the `review` service and the `user` service to ensure user-specific reviews are correctly retrieved and displayed.

By applying mutation testing at both unit and integration levels, developers can detect weaknesses in isolated components and in their interactions, ensuring a comprehensive validation of the application's behavior.

Source Code Overview of Anime Review System

The Anime Review System is a Java Spring Boot application. The core features of the application include:

1. **Role-Based Authentication and Login**:
    - Implements secure access control for both users and admins using JWT authentication.
2. **Anime Review and Rating System**:
    - Users can view reviews and ratings of Anime.
    - Users can add their own reviews and ratings, contributing to the platform's content.
3. **Watchlist Management**:
    - Users can create and manage a personalized watchlist for tracking Anime they are interested in.
4. **Search Functionality**:
    - Allows users to search for Anime by name, making it easy to discover specific content.
5. **Random Anime Suggester**:
    - Recommends random Anime, enhancing the user experience by introducing variety and surprise.

Code Structure

The source code is organized into distinct layers to ensure separation of concerns:

- **Entities Layer**:
    - Contains the core data models (`Review`, `User`, `Role`, `Permission`, `WatchList`) that represent the database tables.
- **DTO Layer**:
    - Includes Data Transfer Objects (`ReviewDTO`, `Token`, `ExtraDTO`) to streamline the exchange of data between layers.

- **Controller Layer**:
    - Handles HTTP requests through the `AnimeController` and `UserController`, enabling the application to expose its core functionalities via REST APIs.

- **Service Layer**:
    - Implements business logic in classes like `AnimeService`, `ReviewService`, and `UserService`.
    - `WatchListService` handles user watchlist functionalities.
- **Repository Layer**:
    - Provides an abstraction for database operations using Spring Data JPA through interfaces like `ReviewRepository`, `UserRepository`, and `WatchListRepository`.
- **Security Layer**:
    - Ensures secure authentication and authorization using JWT tokens, with components such as `JwtAuthFilter`, `JwtService`, and `SecurityConfig`.
- **Configuration Layer**:
    - Contains settings and configurations like `CrosConfig` to handle cross-origin resource sharing.

Test Coverage

The project includes comprehensive test files to validate the functionality of all components, including:

- Controllers: Verifying API endpoints.
- Services: Ensuring the correctness of business logic.
- Entities and DTOs: Testing model consistency and data exchange.
- Security: Validating authentication and role-based access controls.

PIT for Java: A Software Testing Automation Tool

PIT (Pitest) for Java is a mutation testing automation tool designed to evaluate the effectiveness of a test suite in detecting subtle bugs. It is a lightweight and widely used tool in Java applications for improving software quality by simulating faults in the code and determining whether the test cases can detect these faults.

PIT works by generating **mutants** (small changes to the program's source code) and running the test suite against them. Its key advantage is its seamless integration into Java projects and its ability to work with popular testing frameworks like JUnit and TestNG.

Key Features of PIT

1. **Mutation Operators**:
   PIT uses a variety of mutation operators to introduce faults into the code, such as replacing arithmetic operators, logical operators, and relational operators, as well as modifying constants and removing method calls.
2. **Mutation Score**:
   It calculates a mutation score to measure the quality of the test suite
   A high score indicates an effective test suite, while a low score highlights areas needing improvement.
3. **Incremental Analysis**:
   PIT can analyze only the changes made to the codebase, reducing testing time for large projects.
4. **Reports and Insights**:
   PIT generates detailed reports that highlight which parts of the code are well-tested and which require additional attention.
5. **Continuous Integration Support**:
   It integrates with popular CI/CD tools like Jenkins, Maven, and Gradle, making it ideal for automated testing pipelines.

Benefits of Using PIT

1. **Test Suite Evaluation**: It identifies gaps in the test cases by determining their ability to catch injected faults.
2. **Improved Code Quality**: By revealing weak areas in testing, it encourages developers to write more robust and comprehensive test cases.
3. **Realistic Fault Simulation**: The mutations simulate real-world coding errors, ensuring the test suite is capable of detecting them.
4. **Ease of Use**: PIT requires minimal configuration and integrates effortlessly into Java projects.
5. **Efficiency**: PIT's performance-optimized engine ensures efficient mutant generation and testing, even for large codebases.

Use Cases of PIT

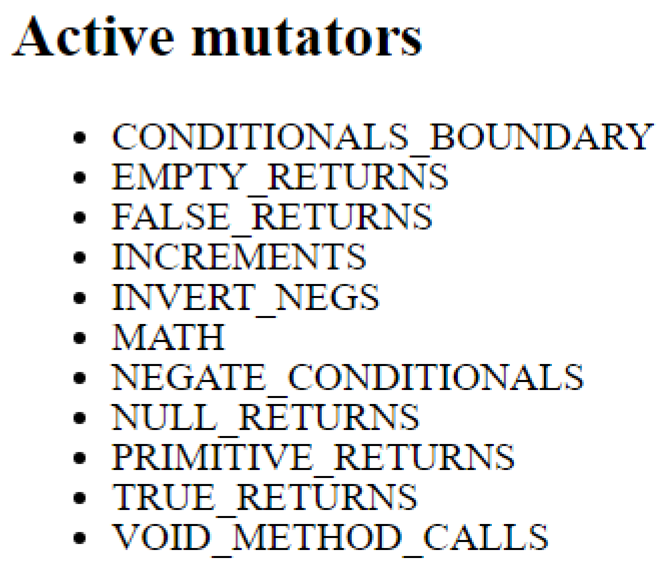
1. **Evaluating Unit Tests**: Assess the thoroughness of test cases at the unit level.
2. **Improving Test Coverage**: Identify untested or poorly tested code.
3. **Regression Testing**: Ensure newly added or modified code does not introduce undetected faults.

PIT for Java is a powerful tool for automated mutation testing, providing actionable insights into test suite effectiveness and helping developers deliver high-quality software. Its simplicity, flexibility, and integration capabilities make it an essential tool for modern software testing practices.

PIT Report for Anime Review System

Mutation Operators:

The active mutators are as follows:



Active mutators

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- TRUE_RETURNS
- VOID_METHOD_CALLS

Mutation Operators used for Unit Testing:

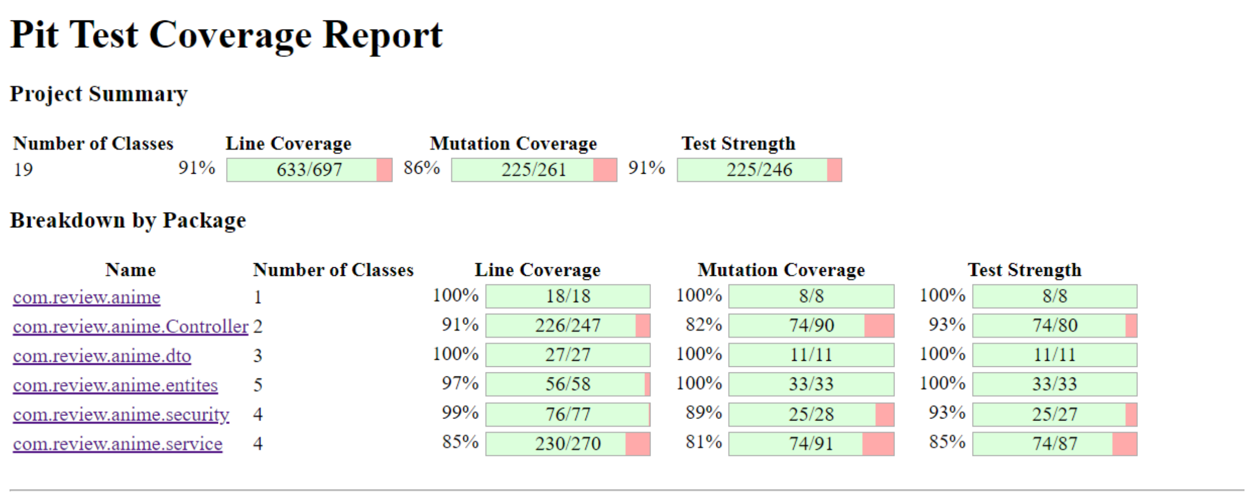
1. Negate Conditionals
2. Conditionals_Boundary
3. Math

Mutation Operators used for Integration Testing:

1. FALSE Returns
2. TRUE Returns
3. NULL Returns

Above are the most common operators used in the testing report, while others have also been used.

Overall Testing Coverage report by PIT:



Pit Test Coverage Report

Project Summary

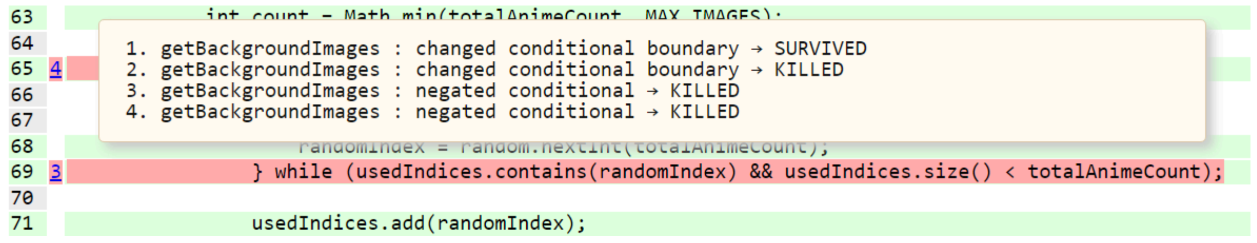
Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
19	91%	633/697	86%	225/261	91%	225/246

Breakdown by Package

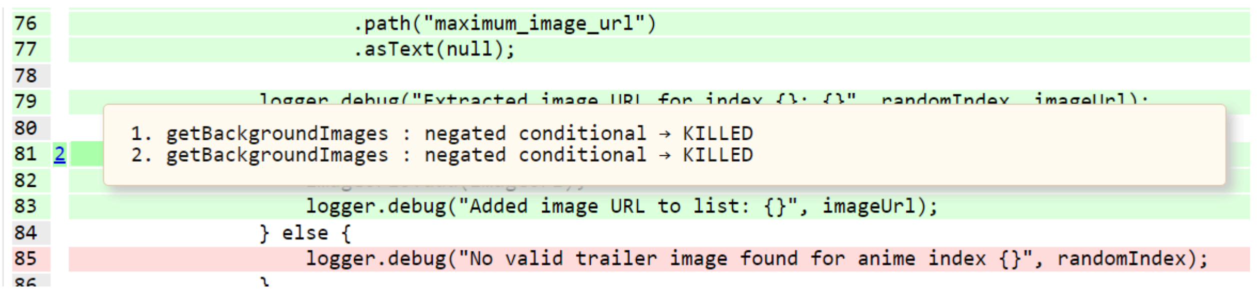
Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
com.review.anime	1	100%	18/18	100%	8/8	100%	8/8
com.review.anime.Controller	2	91%	226/247	82%	74/90	93%	74/80
com.review.anime.dto	3	100%	27/27	100%	11/11	100%	11/11
com.review.anime.entites	5	97%	56/58	100%	33/33	100%	33/33
com.review.anime.security	4	99%	76/77	89%	25/28	93%	25/27
com.review.anime.service	4	85%	230/270	81%	74/91	85%	74/87

Testing Screenshots from PIT Reports

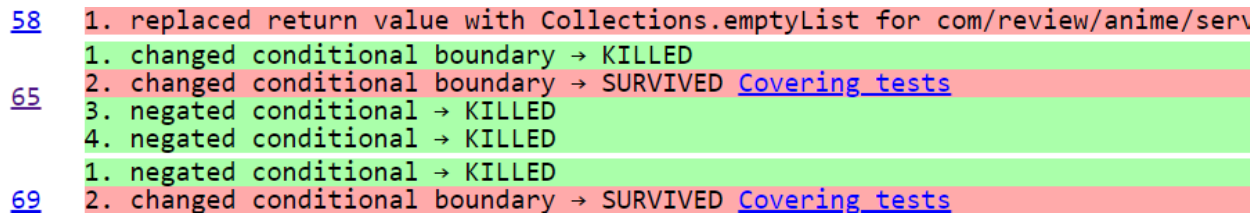
Unit Testing



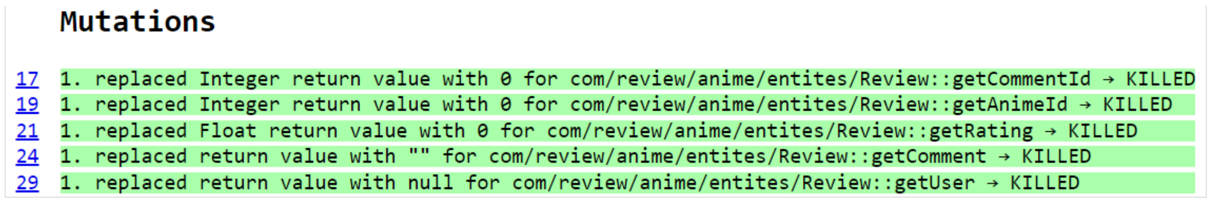
```
            int count = Math.min(totalAnimeCount, MAX_IMAGES);

  1. getBackgroundImages : changed conditional boundary → SURVIVED
  2. getBackgroundImages : changed conditional boundary → KILLED
  3. getBackgroundImages : negated conditional → KILLED
  4. getBackgroundImages : negated conditional → KILLED

                randomIndex = random.nextInt(totalAnimeCount);
            } while (usedIndices.contains(randomIndex) && usedIndices.size() < totalAnimeCount);

            usedIndices.add(randomIndex);
```



```
                        .path("maximum_image_url")
                        .asText(null);

                logger.debug("Extracted image URL for index {}: {}", randomIndex, imageUrl);

  1. getBackgroundImages : negated conditional → KILLED
  2. getBackgroundImages : negated conditional → KILLED

                    logger.debug("Added image URL to list: {}", imageUrl);
                } else {
                    logger.debug("No valid trailer image found for anime index {}", randomIndex);
                }
```

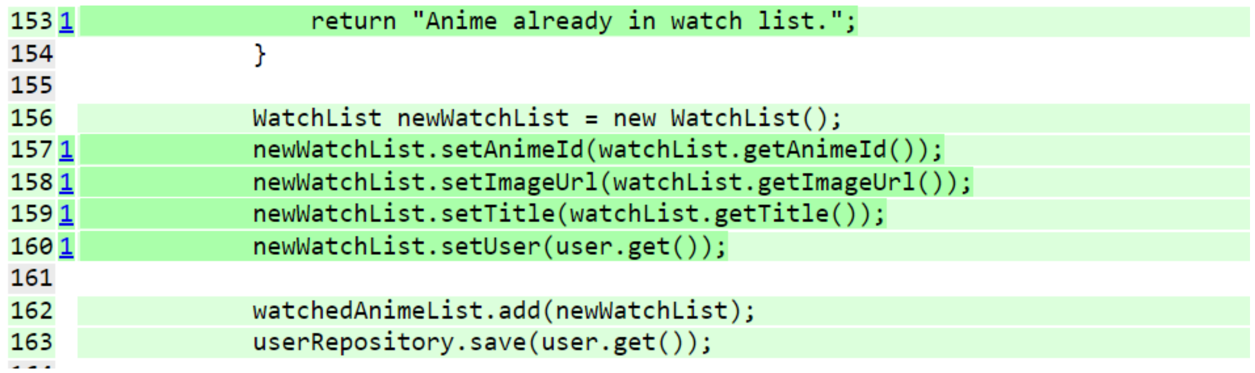


```
58  1. replaced return value with Collections.emptyList for com/review/anime/serv
65  1. changed conditional boundary → KILLED
    2. changed conditional boundary → SURVIVED Covering tests
    3. negated conditional → KILLED
    4. negated conditional → KILLED
69  1. negated conditional → KILLED
    2. changed conditional boundary → SURVIVED Covering tests
```



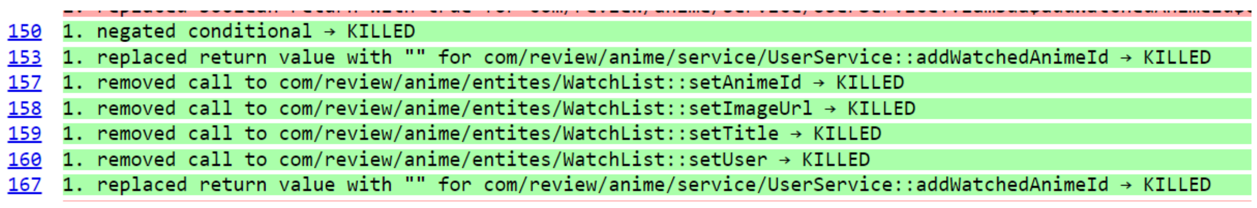
Mutations

```
17  1. replaced Integer return value with 0 for com/review/anime/entites/Review::getCommentId → KILLED
19  1. replaced Integer return value with 0 for com/review/anime/entites/Review::getAnimeId → KILLED
21  1. replaced Float return value with 0 for com/review/anime/entites/Review::getRating → KILLED
24  1. replaced return value with "" for com/review/anime/entites/Review::getComment → KILLED
29  1. replaced return value with null for com/review/anime/entites/Review::getUser → KILLED
```

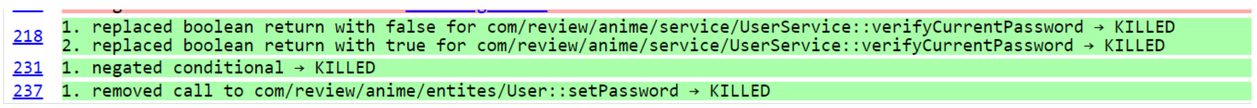


```
                    return "Anime already in watch list.";
                }

                WatchList newWatchList = new WatchList();
                newWatchList.setAnimeId(watchList.getAnimeId());
                newWatchList.setImageUrl(watchList.getImageUrl());
                newWatchList.setTitle(watchList.getTitle());
                newWatchList.setUser(user.get());

                watchedAnimeList.add(newWatchList);
                userRepository.save(user.get());
```

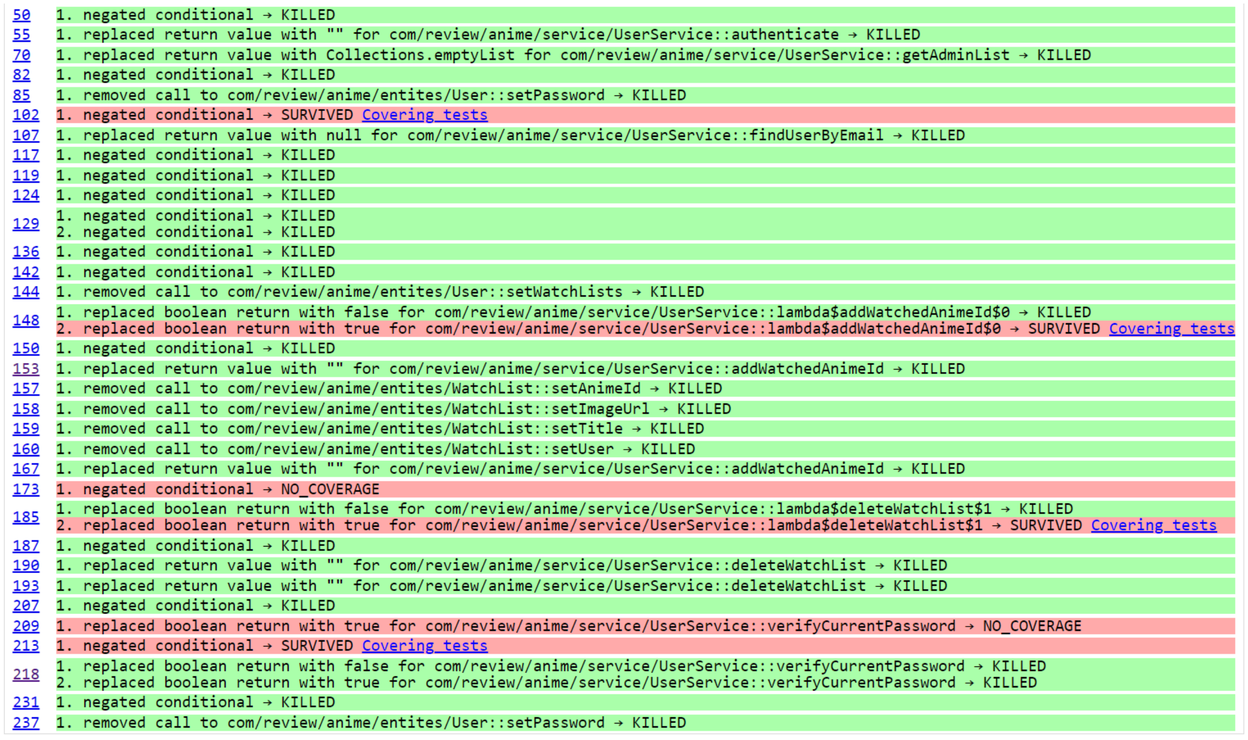
Integration Testing:



```
150  1. negated conditional → KILLED
153  1. replaced return value with "" for com/review/anime/service/UserService::addWatchedAnimeId → KILLED
157  1. removed call to com/review/anime/entites/WatchList::setAnimeId → KILLED
158  1. removed call to com/review/anime/entites/WatchList::setImageUrl → KILLED
159  1. removed call to com/review/anime/entites/WatchList::setTitle → KILLED
160  1. removed call to com/review/anime/entites/WatchList::setUser → KILLED
167  1. replaced return value with "" for com/review/anime/service/UserService::addWatchedAnimeId → KILLED
```



```
218  1. replaced boolean return with false for com/review/anime/service/UserService::verifyCurrentPassword → KILLED
     2. replaced boolean return with true for com/review/anime/service/UserService::verifyCurrentPassword → KILLED
231  1. negated conditional → KILLED
237  1. removed call to com/review/anime/entites/User::setPassword → KILLED
```



```
50   1. negated conditional → KILLED
55   1. replaced return value with "" for com/review/anime/service/UserService::authenticate → KILLED
70   1. replaced return value with Collections.emptyList for com/review/anime/service/UserService::getAdminList → KILLED
82   1. negated conditional → KILLED
85   1. removed call to com/review/anime/entites/User::setPassword → KILLED
102  1. negated conditional → SURVIVED Covering tests
107  1. replaced return value with null for com/review/anime/service/UserService::findUserByEmail → KILLED
117  1. negated conditional → KILLED
119  1. negated conditional → KILLED
124  1. negated conditional → KILLED
129  1. negated conditional → KILLED
     2. negated conditional → KILLED
136  1. negated conditional → KILLED
142  1. negated conditional → KILLED
144  1. removed call to com/review/anime/entites/User::setWatchLists → KILLED
148  1. replaced boolean return with false for com/review/anime/service/UserService::lambda$addWatchedAnimeId$0 → KILLED
     2. replaced boolean return with true for com/review/anime/service/UserService::lambda$addWatchedAnimeId$0 → SURVIVED Covering tests
150  1. negated conditional → KILLED
153  1. replaced return value with "" for com/review/anime/service/UserService::addWatchedAnimeId → KILLED
157  1. removed call to com/review/anime/entites/WatchList::setAnimeId → KILLED
158  1. removed call to com/review/anime/entites/WatchList::setImageUrl → KILLED
159  1. removed call to com/review/anime/entites/WatchList::setTitle → KILLED
160  1. removed call to com/review/anime/entites/WatchList::setUser → KILLED
167  1. replaced return value with "" for com/review/anime/service/UserService::addWatchedAnimeId → KILLED
173  1. negated conditional → NO_COVERAGE
185  1. replaced boolean return with false for com/review/anime/service/UserService::lambda$deleteWatchList$1 → KILLED
     2. replaced boolean return with true for com/review/anime/service/UserService::lambda$deleteWatchList$1 → SURVIVED Covering tests
187  1. negated conditional → KILLED
190  1. replaced return value with "" for com/review/anime/service/UserService::deleteWatchList → KILLED
193  1. replaced return value with "" for com/review/anime/service/UserService::deleteWatchList → KILLED
207  1. negated conditional → KILLED
209  1. replaced boolean return with true for com/review/anime/service/UserService::verifyCurrentPassword → NO_COVERAGE
213  1. negated conditional → SURVIVED Covering tests
218  1. replaced boolean return with false for com/review/anime/service/UserService::verifyCurrentPassword → KILLED
     2. replaced boolean return with true for com/review/anime/service/UserService::verifyCurrentPassword → KILLED
231  1. negated conditional → KILLED
237  1. removed call to com/review/anime/entites/User::setPassword → KILLED
```