Sigma Journal of Engineering and Natural Sciences
Web page info: https://sigma.yildiz.edu.tr
DOI: 10.14744/sigma.2024.00084

Research Article

Automatic unit test generator software by collecting run-time data

Sevdanur GENÇ[1,*]

[1]*Department of Computer Technologies, Taşköprü Vocational School, Kastamonu University, Kastamonu, 37400, Türkiye*

ARTICLE INFO

Article history
Received: 17 November 2022
Revised: 05 January 2023
Accepted: 04 April 2023

Keywords:
Byte Code; Java Agent; Software Testing; The Opcode Parsing Method; Unit Test Generation

ABSTRACT

Quality and productivity needs are considered together in software. For this reason, any existing software should be tested automatically with test automation. Software test automations is automated software testing activities. Automating constantly manually written tests, on the other hand, saves time, reduces error rates, produces better quality software, and reduces costs. This study aims to produce an automatic unit testing framework that is planned to work in run-time on software products. This developed application performs unit test transformations that can respond to the desired test scenarios on the product being studied.. Java agent is used as the basis of all these transformations. All information about the objects, methods, and variables of the sample java classes to be worked on is converted into data in run-time using byte code. During this transformation, information is saved in the database, and unit tests are created automatically through the template engine. Compared to the products developed on automatic unit test generation in the literature, the opcode parsing method was developed for this study. This method reads a byte code at run-time, uses the properties of the java class it belongs to, and automatically creates the unit test class and test methods. The study can also examine different object definitions and conditional and loop structures within a method and produce alternative test scenarios. The automatic unit test scenario produced has been turned into a flexible framework that can encounter minimum errors at run-time.
Considering the scarcity of studies in the field of national software testing; It is thought that the automatic unit test generation product developed within the scope of this study, using byte code, will contribute to the work area.

Cite this article as: Genç S. Automatic unit test generator software by collecting run-time data. Sigma J Eng Nat Sci 2024;42(4):988–1008.

INTRODUCTION

The main purpose of designing the Java programming language, developed by Sun Microsystems and made available in 1995, was to develop a portable, easy-to-learn, general-purpose, platform-independent, object-oriented programming language. The Java compiler converts the source code into java bytecode, which is a platform-independent intermediate language. This code is then processed and run through the java virtual machine on each platform. Java agents are used to flexibly modify the application logic executed by the JVM at this run-time of the code. A java agent is a specially crafted jar file. This file uses the

***Corresponding author.**
*E-mail address: sgenc@kastamonu.edu.tr

This paper was recommended for publication in revised form by Editor in-Chief Ahmet Selim Dalkilic

Instrumentation API to replace the existing bytecode loaded in the JVM. The most important feature of Java agent tools is that they can redefine or modify classes at run-time. They can modify method bodies by redefining their constant and variable properties. They can also change the signatures or inheritance properties of methods.

Quality and productivity needs are considered together in software. For this reason, factors such as a fluent algorithm and strong risk management are needed to test the existing software's compliance with these criteria. In order to fulfill these factors, there are serious responsibilities in the test area. Fast tests and high accuracy of the results are the factors that make a difference in software development. In order to realize this factor, software test automation is needed.

The focus of projects on software is software testing processes. At the end of a successful test process, highly accurate software with the least errors is produced. Studies on software quality in Turkey show that the need for the test-driven software development process and test tools is increasing in our country [1]. In test-driven software development, the target is to write a testable code first, with the scenario belonging to this code, before writing the code that will do the necessary work. After various software development principles design this testable code, if it gives a result with high accuracy, that software has passed the test successfully. If the test results are unsuccessful, it is returned to the beginning, the code is examined, and the problem is tried to be corrected. In software projects, tests called unit tests are written to prove that each unit (class or method) works flawlessly. Unit tests facilitate and accelerate the software development process and ensure that each class and method works correctly.

There are two of the most well-known basic testing frameworks on Java platforms. These are JUnit and TestNG [2]. Both are powerful enough to allow testing in complex test cases on exactly the requested code snippets. Junit is an open-source framework for writing and running repeatable tests. JUnit; runs test data with various test cases to test the expected results from the program. TestNG is more functional and easier to use than Junit. TestNG supports running test cases in parallel on test threads. It also has many features, such as flexible test configurations, detailed analysis of error messages, advanced archiving, and plugin support for editors.

The developer manually codes all these unit tests. Automating the tests is recommended to use the time in a quality manner and speed up the business traffic. Therefore, the speed and accuracy of testing tools are important. For example, keyword-driven scripts have significant advantages. In this approach, the size of the software being tested is important, not the number of tests. This greatly reduces the script maintenance cost and speeds up the implementation of automated tests. At the same time, in order to achieve success in test automation, scripts and data must be reusable. This eliminates repeated tasks and speeds up the implementation of new tests. However, automated tests can also help prevent writing errors compared to manual tests [3].

This study aims to automatically produce unit tests that are planned to work on products in run-time. This software has been developed as a desktop application and can perform unit test transformations on the specified java classes. All these transformations are based on java agents and bytecode. All information about the objects, methods and variables of the sample java classes to be worked on has been converted into data in run-time. During this transformation, information is both saved in the database and unit tests are created automatically through the template engine. However, in this developed study, both the byte code side is analyzed with the improved opcode parsing method and tests are produced on the desired units. At the same time, alternative scenarios were produced for each different use of the objects used in the classrooms. These scenarios can run as soon as the test is applied and perform automatic unit test generation. This automatic unit test generation tool, which can generate unit tests in accordance with the usage differences of each object, has been turned into a flexible framework that can encounter minimum errors at run-time.

Various approaches for automated unit test generation have been presented in the literature. The contributions of the study to the literature are as follows;

1. An opcode parsing method has been developed with the help of java string functions to work during byte code conversions. With this method, values such as objects, variables and input-output parameters against each opcode are distinguished and these data are listed in JSON format. The developed opcode parsing method is open to be developed in line with different needs in the future. In the studies in the literature, limited ready functions of the bytecode API are designed similarly to these operations.
2. Each object in a java class is saved in JSON format using the NoSql database collections so that the values of the variables and input-output parameters can be reused in the unit tests to be created or to assign similar random values to these values. These data are also stored in an archive file in the system. In the studies in the literature, different data storage environments such as XML and oracle have been used.
3. While creating test scenarios, some rules should be observed. The first of these is the alternative cases such as condition and loop structures used in the method. The other is the stage where the mock-stub distinction should be made for different objects used in the method and their value transformations. The user is asked to select a test case through the desktop application developed for these. This selection automatically creates unit tests in the framework structure developed according to the desired scenario. At the same time, the opcode

parsing method developed for mock-stub separation leads to automated unit testing generation software.

4. The assertion structure required for each unit test to be created in JUnit standards is prepared using the FTL template engine. While the application automatically prepares unit tests at run-time, annotations are created according to the test case chosen by the user interface. The use of the FTL template engine has not been found in any automated unit testing generation products in the literature. Instead, different methods have been developed.

RELATED WORKS

Among the different studies in the literatüre published in the last 20 years on unit test generation in software testing applications, the main ones are examined.

Csallner et al. developed an automated test generation tool called JCrasher in their work. This tool can work integrated with Eclipse IDE. After examining the information of the sample java class given to the vehicle for test generation, it is tested with random data. They performed the test with JUnit. The tool can also detect errors that occur during the testing phase [4].

Pacheco et al. developed an automated unit test generation tool called Randoop using JUnit. This tool can create feedback-driven unit testing for object-oriented programs. At the same time, it can catch and archive the errors that occur. It is a tool developed to generate random test data and combine test results [5,6].

Simons et al. developed a unit testing tool for java called JWalk. The study consists of two main stages. First of all, advanced features of a sample class are revealed, and then unit tests are applied systematically. As a result, it can provide information about the status of java test classes. It has also been compared to specialist unit testing applications such as JUnit [7].

Sen developed an application named Cute in the java environment in his study. The application was developed by targeting the C programming language for testing the codes written in the c language. This application works by combining automatic and random test logic. It uses executing symbolic code that helps to overcome distinctive input and restrictive solutions. There are cases where this tool needs to be improved, such as the inability to analyze system calls and solve nonlinear integer equations [8].

Charreteur et al. obtained automated test input for Java bytecode programs using a constraint-based reasoning approach. The method has been developed as a constraint model that allows the bytecode program to be searched backward for each bytecode and solve complex constraints on memory shape. This study, which they named JAUT, is a precedent for studies such as Cute, JTEST and PEX [9].

Fraser et al. developed a test generation tool called EvoSuite in their work. Written in Java, this test generation tool has extensive features. All tests performed with this tool are compared against the desired criteria. Analysis and optimization processes are performed as a result of the comparison [10].

Sakti et al. developed an automated test generation tool called JTExpert that can be used in java programming. The JTExpert tool, which is an executable jar file, takes a java file or java project directory as input and automatically generates a test data package in JUnit format for each tested java class [11].

Tanno et al. developed a hybrid unit test tool called CATG in their work. They used a concept called the concholic test, which dynamically performs the symbolic and concrete inputs [12].

Brill et al. developed an open-source tool called TACKLETEST to create test scenarios at the automatic unit level for java applications. It was developed in the context of application modernization at IBM, but is also used as a general-purpose test creation tool. Overall, it implements a new and complementary way of calculating coverage targets for unit testing through a new white-box combinatorial testing application. This tool establishes a new combinatorial test-based approach for computational scope targets that extensively implement different combinations of parameter types of methods tested at configurable interaction levels [13].

Higo et al. developed a dataset creation tool using automated test creation techniques. They predict that there is a large amount of source code with different implementations of the same functions and that these can be compiled into a dataset useful for various research in software engineering. However, they generate a dataset of functionally equivalent java methods from a source code of about 36 million lines [14].

Lukasczyk et al. developed an automated unit test creation software for the Python programming language named Pynguin. While many researchers focus on static software programming languages such as Java, researchers have focused on Python, the dynamic programming language that has become popular in the last decade. They have developed an extensible test generation framework for Python that generates regression tests with high code coverage. This tool can be expanded to allow researchers to tailor it to their needs and enable future research [15].

Bardin et al. developed an integrated framework for automated test generation in their 2021 study. The aim of the study was to adapt DSE (Dynamic Symbolic Execution), an ATG (Automatic Test Input Generation) technique, to effectively cover a wide test target class derived from the source code of the tested program [16]. In addition, in this study, they did not use model-based testing techniques that cover the characteristics of the tested code. Only certain sequential programs are considered here, and no study related to Bytecode is encountered. At the same time, they have developed a useful framework for test automation that can create tag coverage in the relevant programs.

Arcuri proposed a white-box testing approach where the tested code is fully accessible by the developers. Test

cases are automatically generated using an evolutionary algorithm, such as the MIO algorithm. Tests are rewarded based on code coverage and fault detection measures using this optimization algorithm. The developed approach is used in conjunction with the open-source tool EvoMaster. Experiments were conducted on five different web services containing more than 22,000 lines of code. The applied technique was able to generate test cases that detected 80 faults in the web services. However, the coverage obtained was lower compared to the existing test cases in these projects. A manual analysis of the results found that interactions with SQL databases currently prevented achieving higher coverage [17].

MATERIALS AND METHODS

Logic of Automated Unit Testing

Software testing is an essential element for the development of a software product. Software tests, each test progresses with the right results, reaching a whole so that the software can work in a performance way. Examples are the selection of test data, the test's prerequisites, and the intended and expected results.

The logic of software test automation is to automate handwritten tests and turn it into a tool. Automation means; is the constant repetition of test scenarios. In this case, the personnel who prepare the test scenarios need automatically running software that can run the codes, and these software are called software test automation. In software test automation, all kinds of testable algorithms, methods, and classes have these algorithms. Test modules are written for testing. This is called unit testing.

Test scenarios used in unit tests are first developed and the firmware is coded according to the results of these scenarios. Its purpose is to search for truths and faults in test results. All bugs found by test code developers can be fixed at run-time if they can be fixed. If not, they can help the relevant units by directing the reports of the test results. The test of each unit that is accepted as correct is continued by writing the code of the firmware, so that after each module is completed, each module is integrated and the product is completed.

In this study, a framework structure of automatic software test automation, in which unit tests will be produced, is designed. During the design phase of this structure, structures such as java bytecode and javassist were used.

Java Bytecode

Just like C and C++ compilers are represented by the assembler, java programs are represented by byte code. The byte to be generated by a java compiler is actually the program itself. Bytecode is required to be a solution to Java's problems such as portability and security. Since the Java compiler's output is not executable, it has to use bytecode. Bytecodes are interpreted by the JVM. In this way, bytecodes are well-optimized at run-time. Through bytecode, a java program can be run in many different environments.

The bytecode stream of a method is a sequence of instructions for the JVM. Each instruction consists of a one-byte opcode followed by zero or more operands. The opcode indicates the action to be performed. If more information is required before the JVM can take action, this information is encoded into one or more operands that immediately follow the opcode. Each opcode type has a mnemonic.

The bytecode instruction set is designed to be complex. All instructions are aligned with byte boundaries except for two codes for table creation. The total number of opcodes is small enough, so that byte codes only take up one byte. This helps minimize the size of class files before they are run by the JVM. It also helps to keep the size of the JVM application small. All computations in the JVM are performed on the stack. Therefore, bytecode instructions run primarily on the stack [18].

The working logic of Bytecode; Java bytecode is machine code in .class file format. Bytecode in Java is the command set for the JVM and works similarly to a compiler. A close examination of the bytecode reveals that there are certain opcodes. Some opcodes have letters like a or i in front of them. For example, aload_0 and iload_2. These prefixes represent the types that the opcode has worked with. The prefix a means that the opcode modifies an object reference. The prefix i means that the opcode is processing an integer. Other opcodes; They are used as b for byte, c for char, and d for double. These prefixes provide information about what type of data is being processed.

A stack-based machine is used for the execution of the bytecode by the JVM. Each thread has a JVM stack that stores its data in its frame and turns it into a framebuffer. Each time a method is called, a frame stack is created and the operand stack contains data such as a set of local variables and the run-time of the current class. The local variables array, also known as the local variables table, contains the method's parameters and is also used to hold the values of local variables. Parameters are stored in the directory, starting with index 0. If the structure is for a constructor or dynamic (instance) method, the reference is stored at position 0. Then position 1 gets the first formal parameter and position 2 gets the second formal parameter. For a static method, the first formal method parameter is stored at position 0, and the second at position 1. The size of the array of local variables is determined at compile time, and the number and size of local variables depend on a formal procedure parameter. The operand stack uses the LIFO (Last in First out) method stack to push and pull values. Certain opcode instructions values are passed to the operand stack: others take the operands from the stack, manipulate it, and pass the result. The operand stack is also used to get return values from procedures.

A Java agent is a special java library that can manipulate bytecodes by interfering with applications running on the

JVM using the Java Instrumentation API. Generally, it is prepared as a jar file. Classes that represent Java agents are nothing more than other classes available in the Java API library. But what makes them special is that they follow a certain rule that allows the java code to block any other application running in the JVM. The sole purpose here is simply to make agents that probe or modify the bytecode, This powerful feature goes beyond what a java program normally does. In a way, it can be entered into a program and alter the bytecode or cause havoc. Javassist, a library for editing bytecodes in Java, allows Java programs to define a new class at run-time and modify a class file when loaded into the JVM.

The Opcode Parsing Method

In order to create unit tests automatically, a java class is first defined in the framework. The defined java class is first converted to bytecode file format with the help of java agent. Then used in this format; Using many opcode java string functions such as class name, variables, objects, methods and their input and output parameters, each of them is determined line by line before the bytecode file and then word by word. For example, opcodes such as invokevirtual, invokestatic, ifge, iflt, ifeq, iinc are controlled by using java string functions in the bytecode file and the corresponding parameters are determined and passed to variables. At the same time, Mock-Stub distinction can be made by controlling the parameters given by the opcodes showing the object types defined in the class. All parameters transferred to variables are listed instantly in JSON format and saved as data using NoSql database structure. The recorded data is automatically converted into unit test format instantly with the help of the FTL template engine after all the conversion works are completed.

Another feature that is noted here is that in this advanced method, unit tests are created simultaneously with different test scenarios for mock-stub structures, which are distinguished according to the difference of objects, and saved to the system. The test codes saved in the system can be requested as a printout according to the test scenario the tester selected from the test case section of the menus.

Thanks to the opcode parsing method produced as a flexible framework, necessary code changes can be made for the desired opcode type. It contributes to this project as it is completely open source code.

Advantages and Disadvantages of the Opcode Parsing Method

Previous studies in the literature used meta-heuristic optimization search techniques such as genetic algorithms to automate a testing task [19]. However, no optimization search method or genetic algorithm method was used in this study. Directly, the opcodes of a java class in which bytecodes are generated are parsed using java string methods.

In addition to the limited and ready-made functions offered by the Bytecode API, and the developed opcode decomposition method, the input-output parameters corresponding to the desired opcode can be accessed in the bytecode-converted output. At the same time, since this developed method is open-source code, the users can add or remove desired features according to their needs within each function.

As long as other classes belonging to an object produced in a class are in the same location, it can be automatically processed with the base class. Especially if it is considered for mock-stub applications, no extra action is required by the user and the developed framework detects this itself.

As used in the study of Venkatesan et al. in the literature, a flexible data storage system was needed in this study to store all kinds of data obtained from the opcode parsing method [20]. Since the output parameters are recorded in JSON format after the opcode parsing method, these data can be easily examined in any NoSQL-based database management system.

A total of 5 types of test scenarios were used in the developed project. In the future, test users can increase or improve the number of these test scenarios in line with their needs, with the project being open source.

The application developed for now can perform detailed operations on a given java class. With this method developed in subsequent studies, it is planned to create automatic unit tests according to the features selected by the user of the whole project after the location of a project belonging to many classes that are connected with each other.

In the system developed as test data, random data belonging to numeric data types such as int and double are assigned. In the next study, data selection is planned automatically over random data covering all data types or even a sample data set that can be imported.

About Other Technologies Used

Apart from java byte code, javassist and java agent technologies, other technologies and structures have been used to create automatic unit test generation software. These; code coverage, mock and stub, NoSql, Maven and FTL.

- Code Coverage is a software measurement technique used to measure how many lines of code are executed during automated tests. A code coverage reports generator for Java projects, JaCoCo is a free code coverage library for Java.
- In non-relational databases, there is no integrity among the data and therefore data can be repeated in different ways. This leads to data inconsistency. Changing the same data in different places on the whole system is difficult to manage. Since the SQL interface is not used, it is named No Relation, meaning non-relational. Thus, the expression NoSQL came to the fore. MongoDB was used for this system in this study.
- Developed by Apache, Maven is a JDT (Java Development Tool) or java developer tool. While developing Maven java projects, it creates a standard within

the project. In line with these standards, it simplifies the project development process and enables the creation of documentation effectively. It is a tool that helps to eliminate the dependency on the library and user interface in the project, and provides convenience for the developer in processes such as compilation and reporting. In fact, it is itself a storage unit rather than a tool. There is a chance to run all libraries, plugins and all necessary information about the developed software on the servers.

- FreeMarker Java Template Engine-FTL (Java Template Engine) is a template engine produced by Apache. It is a Java library that can generate text output such as HTML web pages, emails, configuration files and source codes based on templates and changing data. Generally, a general purpose programming language such as java is used to prepare the data. Then it displays the data prepared using FTL templates. It focuses on "how" the data will be presented in the templates and "what" data will be presented outside the template [21].
- The concept of dependency emerges when the software is considered as a whole.For example, the software may be running dependent on a database. This dependency should be paid attention to when creating the test scenario and it should be tested without using this dependency. At this point, mock objects are needed. When writing unit tests, pseudo-objects are created to replace them in order to be able to work independently of real objects. This event is called Mock-Mocking.

Automatic Unit Test Generator Software by Collecting Run-Time Data

In the study, a framework was developed that enables the creation of automated unit tests by collecting data in run-time. The tool in question was developed as a console and desktop application using the Java programming language in the Netbeans environment. Since this study is built on the java programming language, JUnit framework was used to test java-based codes and development was made in this context.

The flow chart of the developed study is given in Figure 1. Developed framework; It consists of 4 components. These,

- The first module is the screen where the .java and .class files, which contain the codes of the Java programming language, are read,
- The second module is the screen where all data consisting of variables, object, methods and class names read simultaneously from the codes is saved to the database with NoSQL,
- The third module is the screen where the java codes are converted to byte code,
- And lastly, it is the screen about creating the automatically generated unit test class file with the help of byte code and incoming data.

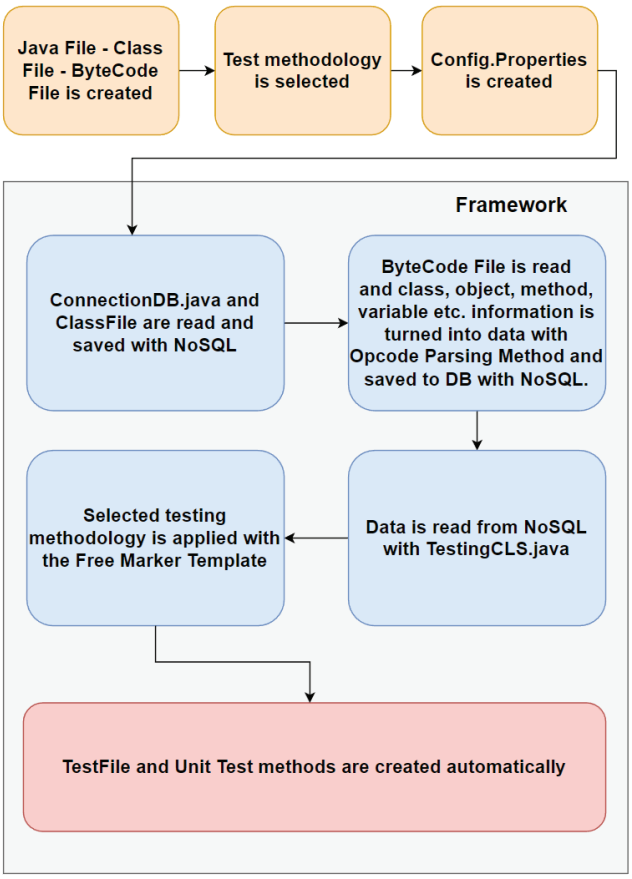


Figure 1. Flowchart of automatic unit test generation software by collecting run-time data.

Identification and Analysis of Scenarios Related to Requirements in Unit Tests

In the study, many scenarios were focused on while creating unit tests. Probability test methods have been created especially for testing methods that contain conditional or loop structures. At the same time, mock and stub structures are also used as needed. A framework has been developed to create automatic unit tests to perform the test stages of all these structures. In this framework, to analyze automatically created test methods and classes, the data used in the software were taken as an example: at the same time, random data were produced and used as a variable. In the sub-headings, each test scenario is examined with code blocks with related simple java class examples.

Alternative States within the Method

When writing a unit test, a single unit test case should be created where a class's method is lean. However, scenarios, where there is more than one situation in the method are very common. In particular, when conditional structures are involved, as in Figure 2, there is a need for a unit test to meet the probability of each condition, and sub-scenarios must be developed for them.

```java
package edu.sdu.sample.project;

public class CalculateCredit {
    public static double CalculateInterest(double principal, double rate){
        if (rate<1.0)
          rate = 1.0;
        return principal/100*rate;
    }
}
```

Figure 2. A method structure with a condition in it.

```java
package edu.sdu.ornek.proje.test;

import edu.sdu.ornek.proje.CalculateCredit;
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculateCreditTest {

    CalculateCredit calCredit = new CalculateCredit();

    @Test
    public void CalculateInterestTest() {
        double calculatedInterest = calCredit.CalculateInterest(100.0,1.20);
        assertEquals(1.20, calculatedInterest, 0.001);
    }

    @Test
    public void CalculateInterestExTest(){
    double calculatedInterest = calCredit.CalculateInterest(100.0, 0.80);
    assertEquals(1.00, calculatedInterest, 0.001);
    }
}
```

Figure 3. Method structures for unit tests of a method with a condition in it.

In this case, as in Figure 3, since there will not be a single unit test that will meet each condition, it is necessary to creating a separate unit test for each condition. Managing such conditional states in a single test method is an anti-pattern.

The reason is that when the test code of the first condition fails, full efficiency cannot be obtained from the test result when the test code of the other conditions is run. In this case, independent tests are expected. As a solution to this situation, different scenarios within the methods are met with separate unit tests.

Different Objects in Method

When writing a unit test, methods can contain different objects. There are sub-methods and objects that these objects in the method depend on. For these sub-methods, the object-generated class is expected to exist ready-made. In summary, the method to be tested depends on the sub-methods of the object. When a normal test code of the code given in Figure 4 is written, a *NullPointerException error* is received.

```java
package edu.sdu.ornek.proje;

public class CalculateCredit {

    public RecordCalculation recCalculation;

    public static double CalculateInterest(double principal, double rate){
        if (rate<1.0)
          rate = 1.0;
        recCalculation.saveCalculation(principal, rate);
        return principal/100*rate;
    }
}
```

Figure 4. A method structure with different objects in it.

```java
package edu.sdu.ornek.proje.test;

import edu.sdu.ornek.proje.RecordCalculation;
import edu.sdu.ornek.proje.CalculateCredit;
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculateCreditTest {

    CalculateCredit calCredit = new CalculateCredit();

    @Test
    public void CalculateInterestTest(){
        calCredit.recCalculation = new RecordCalculation();

        double calculatedInterest = calCredit.saveCalculation(100.0, 1.20);
        assertEquals(1.20, calculatedInterest, 000.1);
    }
}
```

Figure 5. Using the Stub method.

The reason for this NullPointerException error is that the object in the method, namely instance, has not been created. This error can be managed by two situations. The first is the Stub method and the other is the Mock method.

Stub method : It provides ready-made responses to calls made during testing. It usually doesn't respond to anything other than situations written for testing. A class or object structure that implements the methods of the class/object to be called and always returns the desired value. In Figure 5, *calCredit.recCalculation = new RecordCalculation();* line has been added.

This is an option that is not easy to manage. Because there are sub-components of the class that need to be called and an object structure that cannot be easily created. Besides all these, there may be content dependent components such as Spring and EJB. Therefore, due to all these disadvantages, the stub method, which enables to create a counterpart of these objects, is not preferred as much as possible.

Mock method : It is used to ensure everything in the method is correct before returning the correct value. It just tests the behavior and makes sure certain methods are called. Like the stub method, the mock method also acts as a mock. This method has a structure that allows an object to serve as if it were operating normally without its actual existence. That is, empty methods and objects replace real methods and objects. However, the difference is; stub focuses on a testable version of a particular object, while mock focuses on the correctness of everything.

Mockito library is used together with JUnit for mocking. The downside is that there is no integration with the

```java
package edu.sdu.ornek.proje.test;

import edu.sdu.ornek.proje.RecordCalculation;
import edu.sdu.ornek.proje.CalculateCredit;

import org.junit.Test;
import org.mockito.Mockito;
import static org.junit.Assert.*;

public class CalculateCreditTest {

    CalculateCredit calCredit = new CalculateCredit();

    @Test
    public void CalculateInterestTest() {
        calCredit.recCalculation = Mockito.mock(RecordCalculation.class);
        double calculatedInterest = calCredit.CalculateInterest(100.0, 1.20);
        assertEquals(1.20, calculatedInterest,0.001);
    }
}
```

Figure 6. Using the mock method.

respective class. However, unit tests by their nature do not deal with integrations with subunits. Therefore, such tests are expected to be performed within integration tests. In Figure 6, *calCredit.recCalculation = Mockito.mock(RecordCalculation.class);* line is added to mock the object. The existence of a class file belonging to the object is defined as a parameter to the mock method. Accordingly, the content of the save method in that object will no longer be executed.

Annotation-based uses are also available to manage mocking. In Figure 7, the Mockito library shows a method for mocking the object by adding annotations directly to it.

As seen in the figure, if annotations are defined in the test class, the code will become even more readable and concise. Thus, related classes were defined in annotations and an object was produced from them. Afterward, these objects were used in test methods.

Value Conversions on Mocked Objects

Sometimes a real unit test environment cannot be created because methods or objects carry their default values in a mocked object. In this case, AssertionError error is received at the time of test. The meaning of this error indicates that a different result is obtained as a result of comparing the expected and actual values. When the mock method mocks an object, it removes the method bodies inside it. In order to manage this, certain states are assigned to the mocked objects. Given-When-Then standards were used to manage these situations, as seen in Figure 8.

Thus, even if the object is mocked, the desired states can be created and tests can be written. In the Given stage, the variables that are desired to be obtained and to be used are created. The object to be tested is configured. In the When phase, the values and variables are brought together with the configured object in the Given phase, and the code to be tested is processed. In the Then phase, the expected result is passed to the test code.

Design of Automated Unit Test Generation Software

Java-class-bytecode conversions

In the developed framework, first of all, a java class is loaded into the system to create unit tests. This *java file* is converted into a *.class file* with the click of a button, and the .class file is used in almost every part of the developed software. The *JavaCompiler* library is used to compile the class file. If a class that needs more than one java class is read here, class files are read in other classes in the specified location. The developed system stores all the files it reads in the *Iterable* collection list, and then this list is used for conversion operations.

After conversion from Java file to Class file, byte code conversion operations are performed. For this process, *ClassPool, CtClass, CtMethod* and *InstructionPrinter* subclasses in the javassist library are used. The class and method information in the class file being read is kept in the class pool named *classPool*. Afterward, information is drawn from the class pool with *InstructionPrinter* and the results are displayed as a printout.

As seen in Figure 9, in a byte code output, each method in the class is separated between the MethodName lines. Each method starts with line number 0. This method

```
package edu.sdu.ornek.proje.test;

import edu.sdu.ornek.proje.RecordCalculation;
import edu.sdu.ornek.proje.CalculateCredit;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;
import static org.junit.Assert.assertEquals;

@RunWith(MockitoJUnitRunner.class)
public class CalculateCreditTest {

    @InjectMocks
    CalculateCredit calCredit;
    @Mock
    RecordCalculation recCalculation;

    @Test
    public void CalculateInterestTest() {
        double calculatedInterest = calCredit.CalculateInterest(100.0, 1.20);
        assertEquals(1.20, calculatedInterest,0.001);
    }
}
```

Figure 7. Using the mock method with annotation.

```java
package edu.sdu.ornek.proje.test;

import edu.sdu.ornek.proje.InterestRate;
import edu.sdu.ornek.proje.CalculateCredit;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;
import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.*;

@RunWith(MockitoJUnitRunner.class)
public class CalculateCreditTest {

    @InjectMocks
    CalculateCredit calCredit;
    @Mock
    InterestRate intRate;

    @Test
    public void CalculateInterestTest() {
        when(intRate.monthlyRate()).thenReturn(1.20);
        double calculatedInterest = calCredit.CalculateInterest(100.0);
        assertEquals(1.20, calculatedInterest,0.001);
    }
}
```

Figure 8. Using the When-Then Standard when mocking.

```
MethodName : InterestRate
0: dload_0
1: ldc2_w #2 = int 100.0
4: ddiv
5: dload_2
6: dmul
7: dreturn
MethodName : InterestRateEx
0: dload_2
1: dconst_1
2: dcmpg
3: ifge 8
6: dconst_1
7: dstore_2
8: dload_0
9: ldc2_w #2 = int 100.0
12: ddiv
13: dload_2
14: dmul
15: dreturn
```

Figure 9. An example of Bytecode.

returns a value with the *dreturn* keyword, and it also shows that it takes parameters with the keyword *lcd2_w*.

These byte codes are read by the Javassist library and keep the filename and path of the relevant class in parameters such as *ClassPool*, *CtClass* and *CtMethod*. All code lines converted to byte code can be seen on the console screen of the *Netbeans* interface, as well as output as text documents. Another class developed for this framework is *FindObjectsInClasses*. With the help of this class, each byte code is read line by line from the output of the byte code file and these codes are stored in the MongoDB database as NoSQL format. For this, a class called *ConnectionDB* has been created in the framework. In this class, features such as creating a collection and accessing the data in the collection using the methods of the *com.mongodb* library are gathered in a common class. This class also provides archiving of all JSON format data in a text document

The important point here is that the help of java string functions is taken to retrieve the information and save it to the database. This is done by *the opcode parsing method* developed for the study. Variables used in the class, method names, if any, the parameters they have received and the results they have sent, all created objects, Information about used condition and loop blocks are retrieved with the help of string functions, respectively. and saved in MongoDB. In addition to these, information about the opcodes used such as *invokevirtual, getfield, invokestatic, getstatic, ifge, ifle, iflt, ifgt, ifeq, ifne, iinc, if_icmp* and *goto* are also recorded. At the same time, all this information is archived under system files in text document (.txt) format according to the day and time of the transaction. Thus, two collections are used by MongoDB. The first of these is the collection named *byteCoding* seen in Figure 10. Under NoSQL structure, data is listed with key-value structure. The naming of key fields is related to the operations to be performed in some, while in others it reminds of byte code terms. For example, the

```
/* 1 */
{
    "_id" : ObjectId("622e1bb20fb2df14e4b6d6ce"),
    "Execution Id" : "'25'",
    "MethodName" : "'InterestRate'",
    "ClassName" : "'CalculateCredit'",
    "Returned" : "'728.28'"
}

/* 2 */
{
    "_id" : ObjectId("622e1bb20fb2df14e4b6d6cf"),
    "Execution Id" : "'26'",
    "MethodName" : "'InterestRateEx'",
    "ifge12Line" : "< 1.0",
    "ClassName" : "'CalculateCredit'",
    "Returned" : "'1936.69'"
}

/* 3 */
{
    "_id" : ObjectId("622e1bb20fb2df14e4b6d6d0"),
    "Execution Id" : "'27'",
    "MethodName" : "'InterestRateFor'",
    "Loop" : "InterestRateFor",
    "ClassName" : "'CalculateCredit'",
    "Returned" : "'475800'"
}

/* 4 */
{
    "_id" : ObjectId("622e1bb20fb2df14e4b6d6d1"),
    "Execution Id" : "'28'",
    "MethodName" : "'InterestRateIF'",
    "ifle41Line" : "> 7.0",
    "ifge45Line" : "< 9.0",
    "iflt52Line" : ">= 3.0",
    "ifgt56Line" : "<= 6.0",
    "ifle63Line" : "> 1.6",
    "ifgt67Line" : "<= 2.95",
    "ClassName" : "'CalculateCredit'",
    "Returned" : "'4.56'"
}
```


Figure 10. Structure of the collection named *byteCoding*.

key name of *MethodReturnType* stores the value that that method sends as a result, while the key name *ifge45Line* stores the value of the operation to which the if block corresponds.

The second collection name used in MongoDB is *kayitlar*. As seen in Figure 11, all information about classes and methods has been recorded. So much so, that the class and object names of the objects to be mocked are included in this list under the mocking keyword.

Data Migration – Template Extraction

All data archived on MongoDB must be able to communicate with automated unit testing software for operations such as data reading or writing. For this reason, a POJO class has been written to enable communication between the database and the software. All constructor and getter-setter methods suitable for the scenarios are defined together with their variables. The POJO class acts as a data carrier within the framework. The data is stored in two tables in the database. These tables were combined with ids using join and they were communicated with each other. Two different classes are written for the data to be read from the database with these join operations.

The first of these is the *GetItFromMongoDB* class developed for software. The codes of the conditions and loops read in run-time were kept in the *byteCoding* collection. The data read in this collection is directed to *FTL* format to be converted to unit test code with the help of pojo class. Variables for keywords that start with *if* in the *byteCoding* collection; the class name is sent to the pojo class along with the method name and return values. Then, these values are written to the template file named *writeIfTests.ftl* as a unit test method.

The *ReadDataFromDB* class has been created, which pulls data from the MongoDB database. In this class, two collections are connected using join. All relevant variables are both read from the database and checked if there are fields that need to be updated for join operations. All variables are redirected to the framework's main class. At the same time, unit test methods are created by copying the template variables selected for the *FTL* file.

All data pulled from the database is fetched with the bearer class POJO. Together with these data stored in the database, unit tests are automatically prepared in the desired file format using FTL and turned into an output. The content of these outputs are codes belonging to a new class created based on run-time collected data.

The data produced from the parameters used in the java class read to the frame are saved in MongoDB and the data in the database is pulled with FTL


```
/* 1 */
{
    "_id" : ObjectId("622e16570fb2df4654f26bbb"),
    "Execution Id" : "'1:1'",
    "Kaynak" : "'public static double org.brutusin.instrumentation.logging.CalculateCredit3.CalculateInterest(double)'",
    "Baslangic Suresi" : "'Sun Mar 13 19:05:43 EET 2022'",
    "Degiskenler" : "'[484.0]'",
    "ClassName" : "'CalculateCredit3'",
    "MethodName" : "'CalculateInterest'",
    "MethodReturnType" : "'double'",
    "MethodParameters" : "'[double arg0]'",
    "ObjectInClass_Field" : "intRate",
    "ObjectClassName_Method" : "InterestRate",
    "Mocking" : "InterestRate.monthlyRate",
    "ImportMethod" : "org.brutusin.instrumentation.logging.InterestRate",
    "ImportField" : "org.brutusin.instrumentation.logging.CalculateCredit3",
    "Toplam Suresi" : "'277 ms'",
    "Returned" : "'5.808'"
}
```


Figure 11. Structure of the collection named *kayitlar*.

templates. A class that generates random data named *GenerateRandomJavaDataType* has been written in the framework and this random data is used where it is needed in the application. Java string functions are also used in this class, and with the help of these functions, it is determined which data type will be generated randomly for the relevant test class. All generated data is kept in java collection lists and directed for sending to the relevant test class.

RESULTS AND DISCUSSION

Implementation of Automated Unit Test Generation Software

After the back-end codes of the application were completed, the front-end codes were prepared and the unit test code was produced to work in run-time on the scenarios specified in the requirements section.

The application was developed not only as a console environment, but also as a desktop platform. For this, a class called *UnitTestGeneratorGUI* has been designed. The screenshot given in Figure 12 is designed with this class. The class uses the necessary hierarchy for the relevant parts of the framework to work; it contains the functions of all command buttons, lists and text boxes.

Many methods and class types were studied on the developed framework, scenarios created for each alternative situation were tried, and automatic unit test codes were created from all of them, as planned.

Various trial tests and performance analyzes were carried out within the scope of the study. Framework doesn't just work on variables, objects and methods. In addition, additional features such as conditional constructs and loops are also included. With these additional features, studies were carried out with scenarios prepared on the framework, and from all of them a test class file containing them and automated unit test codes that may be suitable as planned.

As seen in the screenshot in Figure 12, the interface consists of two main parts. First, the java code for which the test file is to be prepared is selected with the *Choose* button. Then, in the second part, the data to be sent to the framework is prepared by using the tab object on the screen. The content of the selected java class comes to the *Java Code File* window as a dump. Each code within the class codes will be converted to the bytecode, but any code structure in comments or comment lines are not converted by bytecode. This feature is available in all compilers. In the example in the figure, this class has both conditional and loop constructs.

When the *Get Info* button is clicked, the name of the class and the names of the methods it belongs to will be listed on the right side of the screen. For this, a class named *GetInfoAboutJavaAndByteCodeFile* was designed and run in the background. Here, first of all, the java file, which is

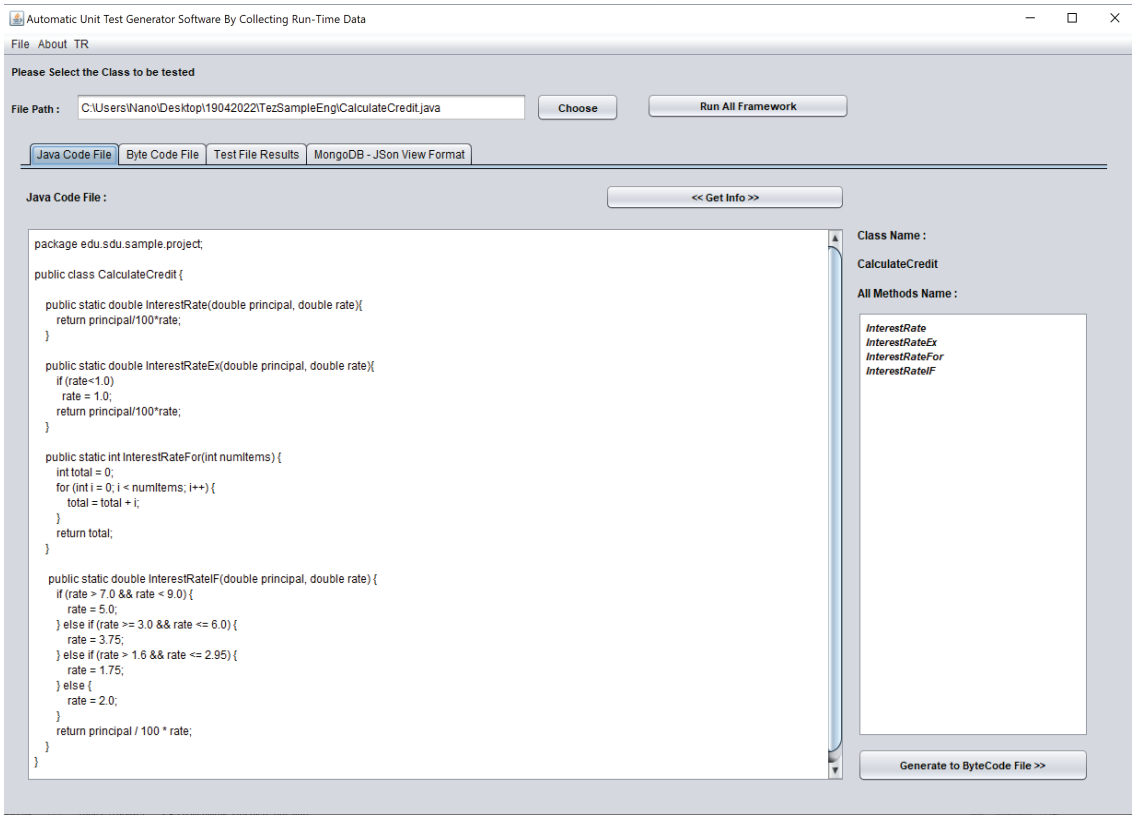


Figure 12. Screenshot of automated unit test generation Framework.

read with the file path, is transformed into a class file by using the *JavaCompiler* library. Along with this conversion, byte code conversion of the relevant class is also performed. After these transformations are completed, the class name of the relevant class and the names of its methods are collected with an array variable and this information is transferred to the list on the right side of the screen. When the *Generate Bytecode File* button is clicked, the *Bytecode File* tab shown in Figure 13 will open.

With the help of the Javassist library, the class file of the relevant example class was created and converted to byte code. Byte codes are created in accordance with the standard determined by the javassist library, separated by method names. The required values from this byte code will be directed into the unit test classes and methods created as a template, and unit tests will be created automatically at run-time. Which test scenario will be used for this, its test case should be selected from the drop-down list at the top of the screen. In addition to normal test methodologies, there are names of test scenarios developed with mock and stub methods. According to the structure of the class opened with the framework, these test scenarios can be selected and the results can be obtained.

By choosing the test case, it is determined with which test scenario the automatic unit test will be generated. Definitions of these scenarios and sample file names are given in the drop-down list with their explanations. After this selection is made, a configuration file is created, such as the lines displayed under the *Contest of the Config File* title in the figure, by clicking the *Generate to ConfigFile* button. In the created configuration file; the names of the java and class files and their path, chosen test method, the name of the java class and the method names it belongs to, along with the file name of the byte code, there is information such as the creation date and time of this configuration file.

Clicking the *Run All Framework* button will pull all the necessary information from this configuration file sequentially. As seen in Figure 14, the command line screen opens and the framework starts working with the information it reads from the *config.properties* configuration file.

Thanks to the Maven structure, it creates three different test class instances that belong to the test codes of the given class according to the data it pulls from the configuration file. At the same time, data is recorded in the database in run-time. The documents belonging to the test classes that come out are both saved in the system as a file and also collectively reflect the results on the command line screen as seen in the figure.

After the command line screen is closed, the *Export Test Results to Lists* button is clicked. The inventories of these unit test files, which are created and then saved in the system, are presented to the user as a screen output in the third

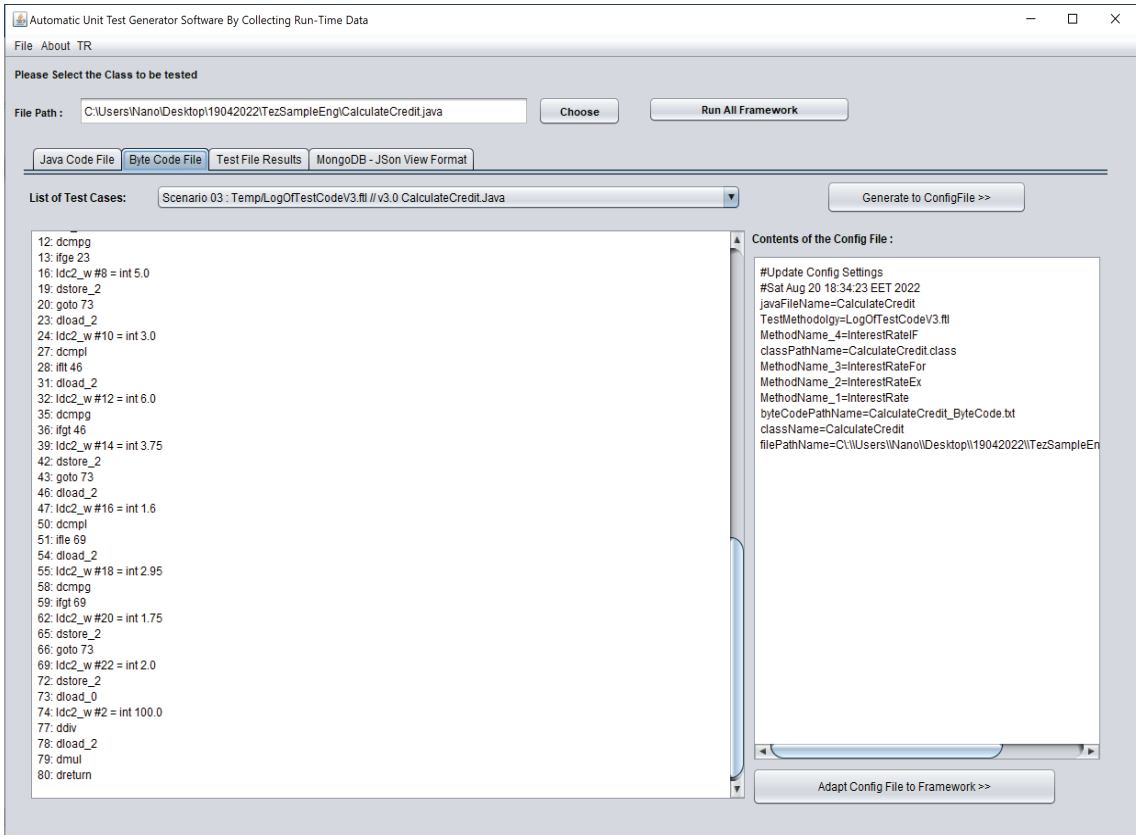


Figure 13. Bytecode conversion of the Java class.

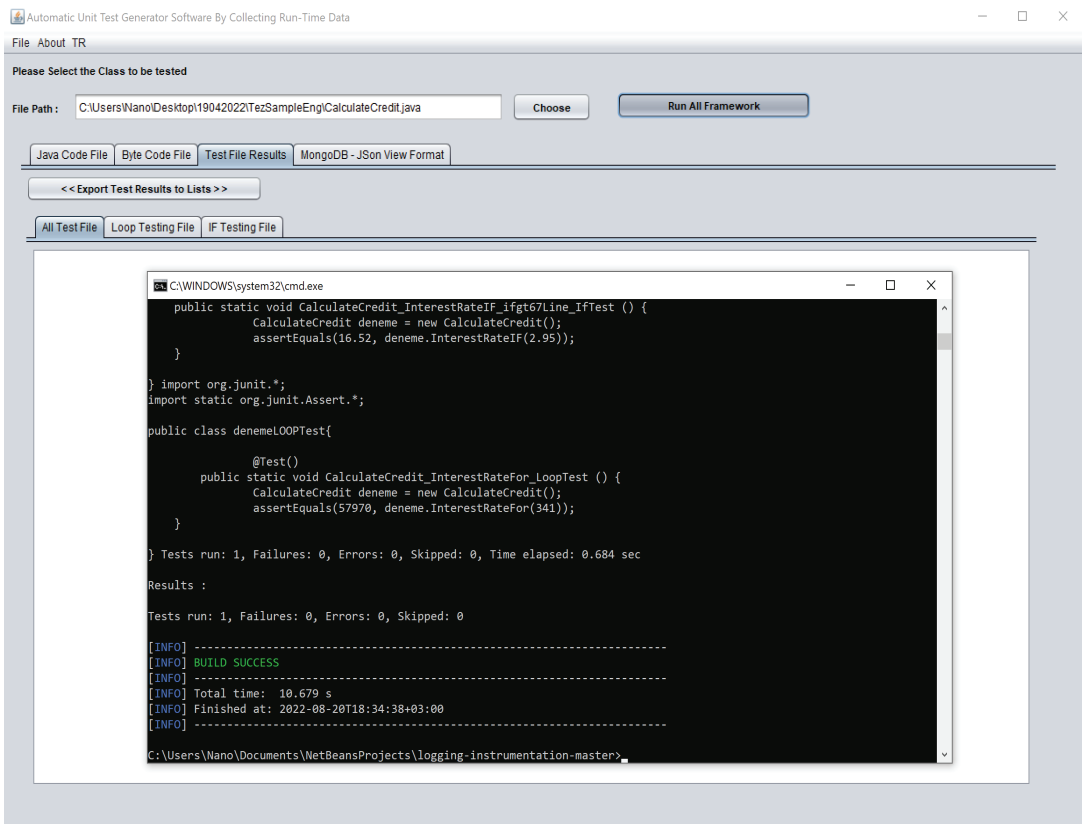


Figure 14. run-time of framework and obtaining test results.

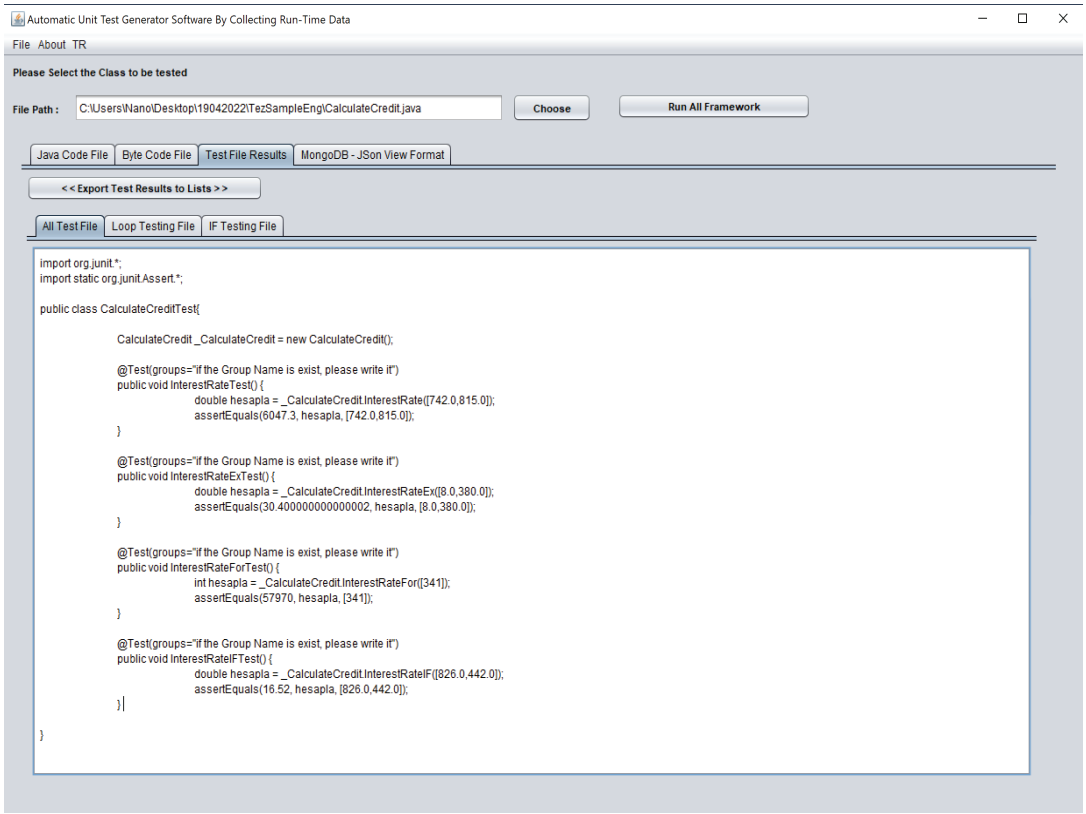


Figure 15. Display of all possible unit test methods.

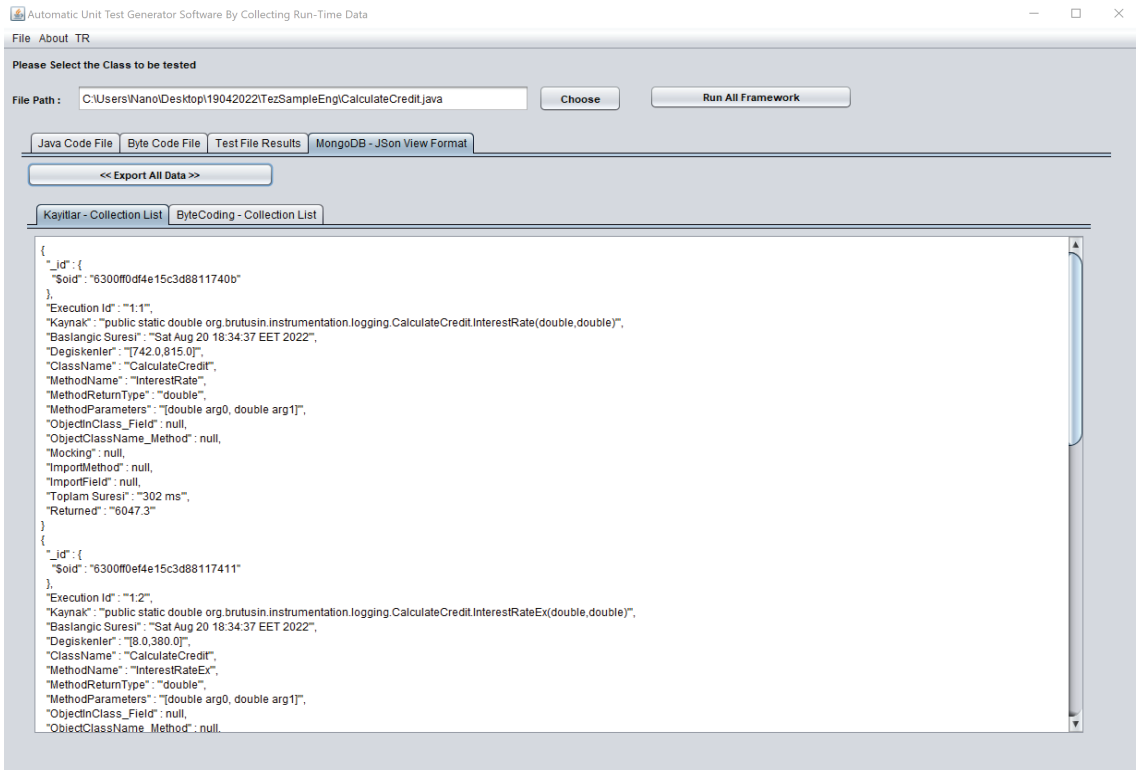


Figure 16. JSon format of *kayitlar* collection in MongoDB.

tab of the application, named Test Results, as seen in Figure 15. There are three sub-tabs on this screen. The first of these tabs is called *All Test File* and this area contains codes for all possible test methods. The second tab contains all possible unit test methods related to loops and the third tab all possible unit test methods related to condition structures.

The fourth tab of the application belongs to the screen named *MongoDB – JSon View Format*. Figure 16 shows two tabs. There are also two NoSQL collection structures in MongoDB software. In the first tab; when the data collected by the framework is logged, class name and method names of the relevant java class, the parameters it takes, values associated with objects, the names belonging to the class names they have, return values from methods, It is the list of the records collection that includes information such as whether there is a mocking or not.

In the second tab; The data of the byteCoding collection can be seen. The analysis of the loop and condition structures of each method in the Java class was made separately, and the summary information of the values related to them was collected and recorded separately.

In Figure 17, a screenshot of the results of the code coverage of each test scenario generated from this software is given.

Realization of Alternative Situations Scenario in Method

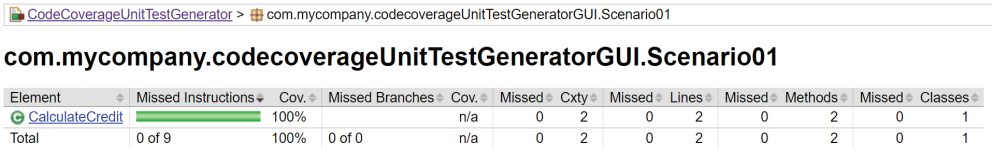
The scenarios given under the title of alternative situations in the method are analyzed under the title of this section. In this case, the code coverage result of the test scenario of a class and a method structure created at a simple level is given in Figure 18. According to this result, it is seen that a successful test is produced with the percentage value of the generated test code.

CodeCoverageUnitTestGenerator — Sessions

CodeCoverageUnitTestGenerator

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario06		68%		n/a	2	4	2	4	2	4	1	2
com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario04		88%		50%	2	5	1	8	1	4	0	2
com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario05		88%		50%	2	5	1	8	1	4	0	2
com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario03		100%		50%	1	5	0	8	0	4	0	2
com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario02		100%		50%	1	3	0	4	0	2	0	1
com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario01		100%		n/a	0	2	0	2	0	2	0	1
Total	11 of 121	90%	4 of 8	50%	8	24	4	34	4	20	1	10

Figure 17. Code coverage results of scenarios.


CodeCoverageUnitTestGenerator > com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario01

com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario01

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
CalculateCredit		100%		n/a	0	2	0	2	0	2	0	1
Total	0 of 9	100%	0 of 0	n/a	0	2	0	2	0	2	0	1

Figure 18. Code coverage result of Scenario1.

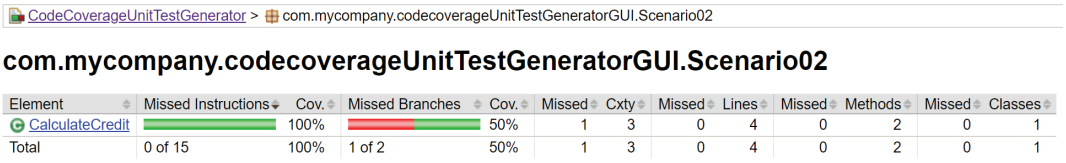
It does not always work with a simple method. There are other alternative method structures in the class. For this reason, thanks to the conditional structure in the method, the test developer resorts to alternative ways while preparing the test scenario. This conditional structure gives the user a kind of guidance on whether the method to be written unit test will work correctly. The result of the code coverage of the scenario of the output given by the automatic unit test creation software is given in Figure 19. According to this result, it is seen that a successful test is produced with the percentage value of the generated test code. Because the condition structure is used in this scenario, the percentage value in the missed branches column has changed.

To summarize this section's analysis results of this section; objects used in a method, loops, and conditional statements are checked and recorded at the time of operation with their relevant data and unit test code generated. However, when there is more than one method in a class, all the values in each method are checked and solutions are developed for each of them.

Realization of Different Objects Scenario in Method

The scenarios given under the title of different objects in the method are analyzed under the title of this section. In this case, different objects are checked within the method as priority. For this, scenarios are realized by creating objects from classes that have interdependencies. For this scenario in the method, a unit test was prepared with two different methods. These methods are realized with the use of mock and stub objects. As can be seen in Figure 20, the result regarding the code coverage of the scenario obtained using the stub method has been obtained.

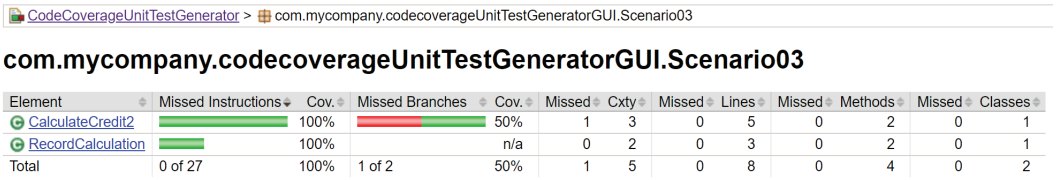
In another scenario, Mockito library is used together with JUnit. With the addition of this library, the object is mocked up. In Figure 21, the code coverage result of the fourth scenario is given. Here, after mocking, both the


CodeCoverageUnitTestGenerator > com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario02

com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario02

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
CalculateCredit		100%		50%	1	3	0	4	0	2	0	1
Total	0 of 15	100%	1 of 2	50%	1	3	0	4	0	2	0	1

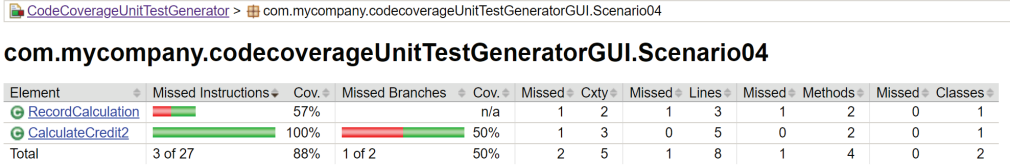
Figure 19. Code coverage result of Scenario2.


CodeCoverageUnitTestGenerator > com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario03

com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario03

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
CalculateCredit2		100%		50%	1	3	0	5	0	2	0	1
RecordCalculation		100%		n/a	0	2	0	3	0	2	0	1
Total	0 of 27	100%	1 of 2	50%	1	5	0	8	0	4	0	2

Figure 20. Code coverage result of scenario3.


CodeCoverageUnitTestGenerator > com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario04

com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario04

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
RecordCalculation		57%		n/a	1	2	1	3	1	2	0	1
CalculateCredit2		100%		50%	1	3	0	5	0	2	0	1
Total	3 of 27	88%	1 of 2	50%	2	5	1	8	1	4	0	2

Figure 21. Code coverage result of Scenario4.

CodeCoverageUnitTestGenerator > com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario05

com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario05

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
RecordCalculation		57%		n/a	1	2	1	3	1	2	0	1
CalculateCredit2		100%		50%	1	3	0	5	0	2	0	1
Total	3 of 27	88%	1 of 2	50%	2	5	1	8	1	4	0	2

Figure 22. Code coverage result of scenario5.

percentage value of *RecordCalculation* and the value of *CalculateCredit2* appear low.

Another use of the mock method is with annotations. Mockito directly marks the object with annotations and performs the mocking process. In Figure 22, the code coverage result of the fifth scenario is given. Here, after mocking, both the percentile value of *RecordCalculation* and the value of *CalculateCredit2* appear low.

Another feature that needs attention in mocked objects is the value transformations of the objects. Methods and objects in the mocked object have their default values. For this reason, baseline values cannot be generated in the unit test environment from time to time. In order to overcome this, *Given-When-Then* standards are followed. Figure 23 shows how this situation was resolved with a test scenario.

In Figure 24, the code coverage result of the sixth scenario is given. Here, the existence of the mocked object and

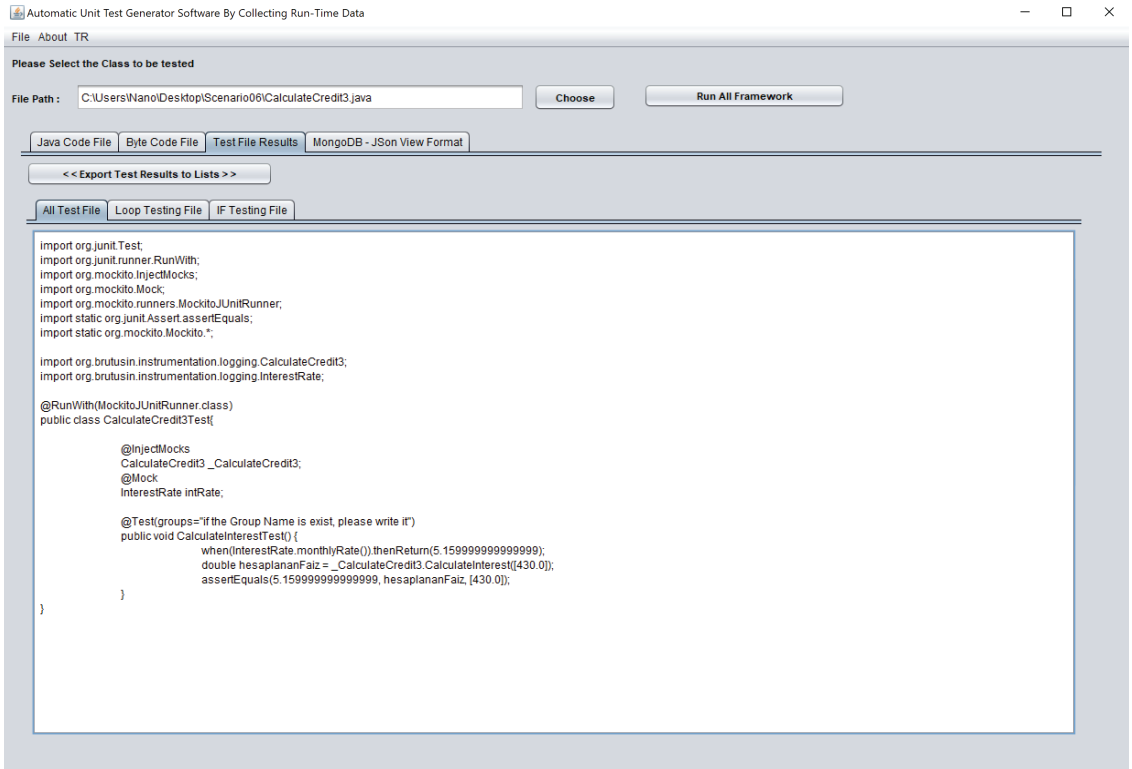


Figure 23. Realization of scenario 6.

CodeCoverageUnitTestGenerator > com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario06

com.mycompany.codecoverageUnitTestGeneratorGUI.Scenario06

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
InterestRate		0%		n/a	2	2	2	2	2	2	1	1
CalculateCredit3		100%		n/a	0	2	0	2	0	2	0	1
Total	5 of 16	68%	0 of 0	n/a	2	4	2	4	2	4	1	2

Figure 24. Code coverage result of scenario6.

the *InterestRate* class can be interpreted from the falling percentage rate.

Thus, it is shown how six important scenarios are resolved in automated unit test creation.

Examination of the Developed Framework in Real-World Unit Test Case Studies

Experiments were conducted on a computer with an i7 2.50 GHz CPU and 16.0 GB memory, running on Windows 10 with JDK 11. For the developed framework, real-world unit test case studies are also reviewed. Multiple experiments were conducted for each test scenario given in the study. For example, for each test case, five independent and simple Java classes were selected from the SF110 Corpus of Classes [22].

SF110 classes collection is an open source repository of 100 sample java projects. Within this repository, unit tests of each project are also included. Line spacing of each java class is generally; It is 10 - 100, 100 - 1000 and 1000 - 10000. These experiments included code blocks and class structures that could yield different CPU and memory results, such as a simple test method and a comprehensive test method like mock. In particular, data obtained from bytecode files were transformed using an *opcode parsing method* in this test production framework. Therefore, the number of lines of code (LOC) for this was also added to the table. The results such as CPU, memory, time, number of lines of code in Java class files and number of lines of code in bytecode files, which are approximate value ranges for all examples, are shown in Table 1.

In this study, test scenarios were determined and test cases were classified in a table. Test cases were examined in six different categories: test case of a simple java method (*TC - 01*), test case created for alternative situations in a java method (including complex methods with conditional and/or loop structures) (*TC - 02*), test case created for different objects in a java method using the Stub method (*TC - 03*), test case resolved with the Mockito library (*TC - 04*), test case resolved with the Mock method's annotations (i.e. MockitoJUnitRunner) (*TC - 05*), and test case created for value conversions performed for fake objects using the Given-When-Then standard (*TC - 06*).

According to the results, the conversion time varies with an increase in the number of code lines for bytecode and Java classes. Additionally, there is an increase in difficulty levels of test scenarios from *TC - 01* to *TC - 06*, which also affects the conversion time. Furthermore, the processing of different classes, object creation, mock-stub, loop and conditional structures were considered in the analysis.

The comparison of the automatic unit test creation software and the processes performed with it by collecting the generated run-time data with other widely used automatic unit test creation tools in the literature is discussed in the discussion and conclusion section.

The application of this developed study has been published at *https://github.com/SevdanurGENC/Nano-Automatic-Unit-Test-Generator*.

Table 1. Approximate value ranges obtained from The Opcode Parsing Method when considering Bytecode LOC.

Test Case	Bytecode LOC	Java Class LOC	Time (Sec)	CPU (%)	Memory (Mb)
	5 – 200	*10 – 100*	*10 – 13*		*15 – 18*
TC – 01	*200 – 2100*	*100 – 1000*	*13 – 16*	*15 – 20*	*18 – 21*
	2100 – 22000	*1000 – 10000*	*16 – 20*		*21 – 25*
	5 – 230	*10 – 100*	*12 – 16*		*16 – 21*
TC – 02	*230 – 2250*	*100 – 1000*	*16 – 19*	*17 – 22*	*21 – 26*
	2250 – 23200	*1000 – 10000*	*19 – 22*		*26 – 33*
	5 – 250	*10 – 100*	*13 – 17*		*19 – 25*
TC – 03	*250 – 2340*	*100 – 1000*	*17 – 22*	*17 – 25*	*25 – 33*
	2340 – 24000	*1000 – 10000*	*22 – 25*		*33 – 39*
	5 – 270	*10 – 100*	*15 – 18*		*22 – 31*
TC – 04	*270 – 2400*	*100 – 1000*	*18 – 24*	*20 – 30*	*31 – 39*
	2400 – 27900	*1000 – 10000*	*24 – 29*		*39 – 43*
	5 – 300	*10 – 100*	*16 – 19*		*25 – 35*
TC – 05	*300 – 2800*	*100 – 1000*	*19 – 28*	*23 – 39*	*35 – 49*
	2800 – 30000	*1000 – 10000*	*28 – 37*		*49 – 61*
	5 – 320	*10 – 100*	*17 – 25*		*29 – 41*
TC – 06	*320 – 2950*	*100 – 1000*	*25 – 32*	*23 – 42*	*41 – 55*
	2950 – 31500	*1000 – 10000*	*32 – 41*		*55 - 73*

CONCLUSION

In this study, an application has been developed that collects data at run-time with the help of Java Agent, stores the collected data in the NoSql database and transforms this data into unit test using a JTL template engine. The studies conducted and the developed tool use a different structure compared to previous studies in the literature in various aspects.

A structure has been created to respond to all possible test scenarios mentioned in the study. When the resulting test classes are run in JUnit, results about which class is being tested can be easily obtained. At the same time, it was observed that successful results were obtained with code coverage. Through this framework, which is easy to use for users, unit tests are created automatically in a very short time.

Recent studies show that each test tool prepared for studies is intended to generate test scenarios healthily automatically. While most of the studies were prepared with the Java programming language, programming languages such as C/C++ and C# make up the rest. Many of these are desktop applications, while the rest are developed as web applications or plug-ins. Usually, these test cases are randomly generated to check the running of the programs. For the development of random tests and their derivatives, techniques based on dynamic symbolic execution are mostly preferred.

Pex, one of the important studies in the literature, is a tool developed for unit testing of C# code. It generates test inputs with different parameters for test scenarios by using dynamic symbolic execution techniques. It also produces results based on the return values of the methods. However, it is limited to classes that require complex method arrays. Randoop and EvoSuite can be given as examples of other important studies in the literature made in the Java programming language. Randoop is known for its ease of use, but unlike EvoSuite, it cannot test complex code structures without guidance. It also aims to produce compact test cases with high code coverage. When using code coverage, a common systematic approach is to select a coverage target (for example, a control flow) at a time and generate a test case that implements that specific objective. They developed this technique by working with the bytecode API. In this study, each control flow of the condition and loop blocks is controlled by the opcode parsing method developed on the bytecode. Separate automatic unit test methods were created with solutions suitable for all conditions. At the same time, all objects defined within the Java class are individually determined by the bytecode. During unit test generation, it is decided whether to use the mock or stub method in accordance with the selected test case while transferring these objects. While preparing the output of the code of the unit test, notation operations are created automatically according to the chosen test case.

Both TestFul and eToc tools used a search-based approach aimed at creating JUnit test cases to maximize structural coverage. However, eToc has not been updated for several years. Therefore, it does not include the latest developments for generating test data. On the other hand, TestFul differs from EvoSuite in many critical details and does not have a fully automated feature. For example, TestFul requires manual editing of XML files for each tested class. EvoMaster, an automatic test generation RESTful API, developed using evolutionary algorithm and optimization techniques such as the MIO algorithm. It can export test files in JSON format and can be used in integration with software such as EvoMaster. Within the scope of this study, both the data used in the Java class imported to the application and random data that have a similar approach to these data are produced and used as parameters in the methods required for unit testing. All these operations are recorded at run-time both in a backup file to the system and in a NoSql collection in JSON structure.

Charreteur et al., in their work where they used byte code, they used the limited memory variable method in the java virtual machine. Their application named JAUT, which tests input generation at the bytecode level, performs constraint-based test input generation from Java bytecode. Therefore, it is mainly associated with other works named JPF, Cute and Pex. Unlike these three tools, JAUT performs backward discovery, i.e. it starts from a target bytecode location and discovers a suitable path to step-by-step input. In fact, JPF, Cute, and Pex rely on forward symbolic execution, which involves symbolically evaluating instructions along a path in the same order as execution. Within the scope of this study, a system that converts byte codes with opcode parsing method has been prepared and a different perspective has been brought to these studies. Each opcode line obtained after the conversion of the Java class to java byte code was analyzed in turn, and the object, variable or input-output parameters of the opcode, if any, were determined. These values were then used in unit test generation.

On the subject of Assertion, Randoop allows descriptions of the source code to specify the observer methods to be used to create the annotation. Orstra generates assertions based on observed return values and object states, and adds assertions to check future work against these observations. While such approaches can be used to produce efficient objects, they do not serve to determine which of these assertions are actually useful, and so such techniques can only be checked in regression testing. In contrast, the μTest tool uses mutation testing to select an effective subset of assertions via EvoSuite. Within the scope of this study, the case can be selected from a list of the test scenario optionally from the user through the application. In accordance with this test case, annotations are turned into unit tests while turning them into outputs via FTL.

It is thought that the tool developed within the scope of this study will also take an important place in the national software testing field. The developed tool is

intended to be actively used in unit tests to be carried out by software testers. This tool, which can respond to the most basic test scenarios in its current form, has a structure that can be developed about how it should behave in much more advanced scenarios, since it has a bytecode-based framework. One of the biggest reasons for this is that Java has an open source system. Different modules can also be developed for this framework that will translate other relevant bytecodes in future test scenarios that may be required.

With the development of the automatic unit test creation software, which is targeted as a domestic product, automatic unit tests are created after the opcode conversion processes are carried out. This work currently works for a single java file defined, excluding linked classes of objects generated within the class. In future studies, after adding multiple Java files as project integrity to the framework, generating possible automatic unit tests according to the specified test scenarios is planned. In addition, the latest developments will be compared with all other examples given for SF110 corpus of classes [23]. At the same time, random test data within the framework is currently only performed for numerical data types. Another goal of the study is to both replicate these data types and perform random data assignments from a test data set that can be exported by the user. For this, it is planned to use fuzz testing methods.

AUTHORSHIP CONTRIBUTIONS

Authors equally contributed to this work.

DATA AVAILABILITY STATEMENT

The authors confirm that the data that supports the findings of this study are available within the article. Raw data that support the finding of this study are available from the corresponding author, upon reasonable request.

CONFLICT OF INTEREST

The author declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

ETHICS

There are no ethical issues with the publication of this manuscript.

REFERENCES

[1] Damar M, Özdağoğlu G, Özdağoğlu A. Software quality and standards on a global scale: Trends in the literature from scientific and sectoral perspective. Alphanumeric J 2018;6:325-348. [CrossRef]

[2] Felice S. JUnit Vs TestNG: Differences between JUnit and TestNG. Available at: https://www.browserstack.com/guide/junit-vs-testng. Accessed on Jun 26, 2024.

[3] Graham D, Fewster M. Software test automation: effective use of test execution tools. Boston: Addison-Wesley Professional; 1999.

[4] Csallner C, Smaragdakis Y. JCrasher: An automatic robustness tester for Java. Softw Pract Exp 2004;34:1025-1050. [CrossRef]

[5] Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-directed random test generation. In proceedings of the 29th International Conference on Software Engineering; 2007: Minneapolis, MN, USA. IEEE; 2007. pp.75-84. [CrossRef]

[6] Pacheco C, Ernst MD. Randoop: Feedback-directed random testing for Java. Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2007:815-816. [CrossRef]

[7] Simons AJH. JWalk: A tool for lazy, systematic testing of java classes by design introspection and user interaction. Autom Softw Eng 2007;14:369-418. [CrossRef]

[8] Sen K, Marinov D, Agha G. Cute: A concolic unit testing engine for C. Available at: https://www.cs.columbia.edu/~junfeng/08fa-e6998/sched/readings/cute.pdf. Accessed Jun 26, 2024.

[9] Charreteur F, Gotlieb A. Constraint-based test input generation for java bytecode. In proceedings of the 21st IEEE International Symposium on Software Reliability Engineering; 2010 Nov; San Jose, CA, USA. IEEE; 2012. [CrossRef]

[10] Fraser G, Arcuri A. 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; 2011 Sept; New York, United States. 2011. pp. 416-419. [CrossRef]

[11] Sakti A, Pesant G, Gueheneuc YG. Instance generator and problem representation to improve object oriented code coverage. IEEE Trans Softw Eng 2015;41:294-313. [CrossRef]

[12] Tanno H, Zhang X, Hoshino T, Sen K. TesMa and CATG: automated test generation tools for models of enterprise applications. In proceedings of the 37th IEEE International Conference on Software Engineering; 2015 May 16-24; Florence, Italy. IEEE; 2015. pp. 717-720. [CrossRef]

[13] Tzoref-Brill R, Sinha S, Abu Nassar A, Goldin V, Kermany H. TackleTest: A tool for amplifying test generation via type-based combinatorial coverage. Available at: https://research.ibm.com/publications/tackletest-a-tool-for-amplifying-test-generation-via-type-based-combinatorial-coverage. Accessed on Jun 26, 2024.

[14] Higo Y. Constructing dataset of functionally equivalent Java methods using automated test generation techniques. Available at: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4543198. Accessed on Jun 26, 2024. [CrossRef]

[15] Lukasczyk S, Fraser G. Pynguin: Automated unit test generation for python. Available at: https://arxiv.org/abs/2202.05218. Accessed on Jun 26, 2024.

[16] Bardin S, Kosmatov N, Marcozzi M, Delahaye M. Specify and measure, cover and reveal: A unified framework for automated test generation. Sci Comput Program 2021;207:102641. [CrossRef]

[17] Arcuri A. RESTful API automated test case generation with EvoMaster. ACM Trans Softw Eng Methodol 2019;28:1-37. [CrossRef]

[18] Venners B. Bytecode basics : A first look at the bytecodes of the Java virtual machine. Available at: https://www.infoworld.com/article/2077233/bytecode-basics.html?page=2. Accessed on Jun 26, 2024.

[19] McMinn P. Search-based software testing: past, present and future. In proceedings of the IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops; 2011 Mar 21-25; Berlin, Germany. IEEE; 2011. pp. 153-163. [CrossRef]

[20] Venkatesan P, Rozario RG, Fiaidhi J. Junit framework for unit testing. Available at: https://www.techrxiv.org/doi/full/10.36227/techrxiv.12092259.v1. Accessed on Jun 26, 2024.

[21] FreeMarker. What is a Apache FreeMarker? Available at: https://freemarker.apache.org/index.html. Accessed on Jun 26, 2024.

[22] Evosuite. SF110 corpus of classes. Available at: https://www.evosuite.org/experimental-data/sf110/. Accessed on Jun 26, 2024.

[23] Fraser G, Arcuri A. A large-scale evaluation of automated unit test generation using evosuite. Available at: https://www.evosuite.org/wp-content/papercite-data/pdf/tosem_evaluation.pdf. Accessed on Jun 26, 2024.