IndividualReport

Developed by Rui Neto (1230211)

This folder includes all artifacts developed for the First Part of QSOFT Project.

It is structured as follows:

Contents

List of Figures

1. Introduction

The present report is the outcome of the practical work proposed in the course subject QSOFT of the first year of the Master's in Software Engineering at ISEP.

It is also the result of a comprehensive test of the `Group` aggregate in a Web App designed for Personal Finance Management, that can be found here, to see if it is reusable in the context of another application.

In this application, an individual is characterized by a name, address, birthdate, birthplace, maternal and paternal details, as well as siblings. Individuals can congregate into **Groups** (e.g., families), each having administrators, a description, a creation date, and a Ledger. It is compulsory to record the financial transactions of both individuals and Groups in a Ledger.

The primary objective here is to test the talked before Group aggregate to see if the application is reusable within the scope of another application.

To enhance the understanding of the group's objectives, we present the following domain model diagram. In this diagram, the class highlighted in red represents the aggregate under evaluation.

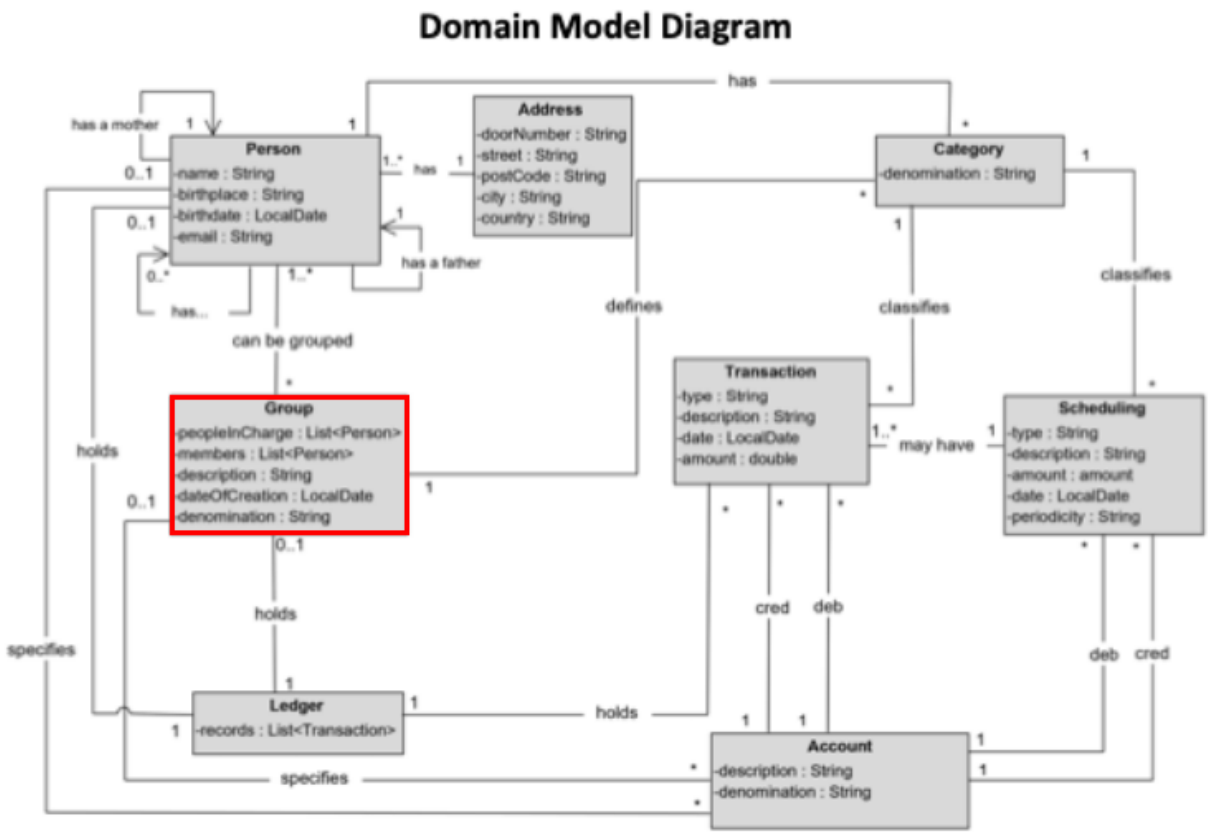

Figure 1- Domain Model

1.1. Sequence diagrams of Group aggregate

In the dynamic landscape of software development and system analysis, the ability to comprehend and communicate complex interactions is paramount. One powerful tool that facilitates this understanding is the sequence diagrams.

In the next steps, it will be possible to see sequence diagrams for creating a group and for adding a person to a group.

1.1.1. Add Person to Group

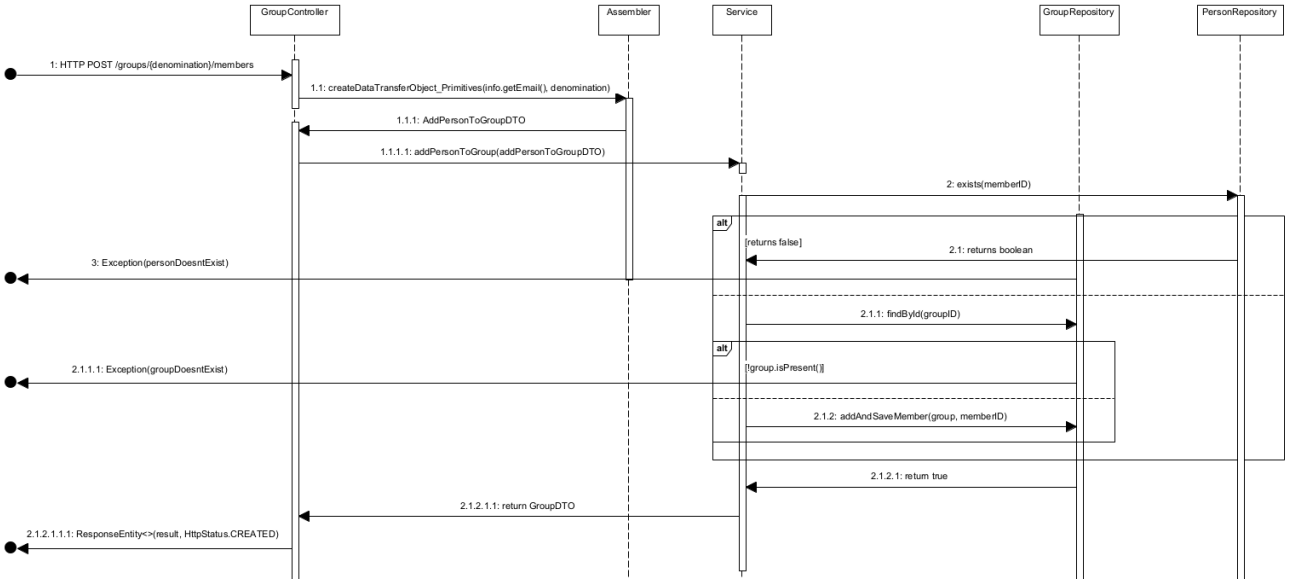

Figure 2- Sequence Diagram - Add Person to Group

1.1.2. Create Group as Person in charge

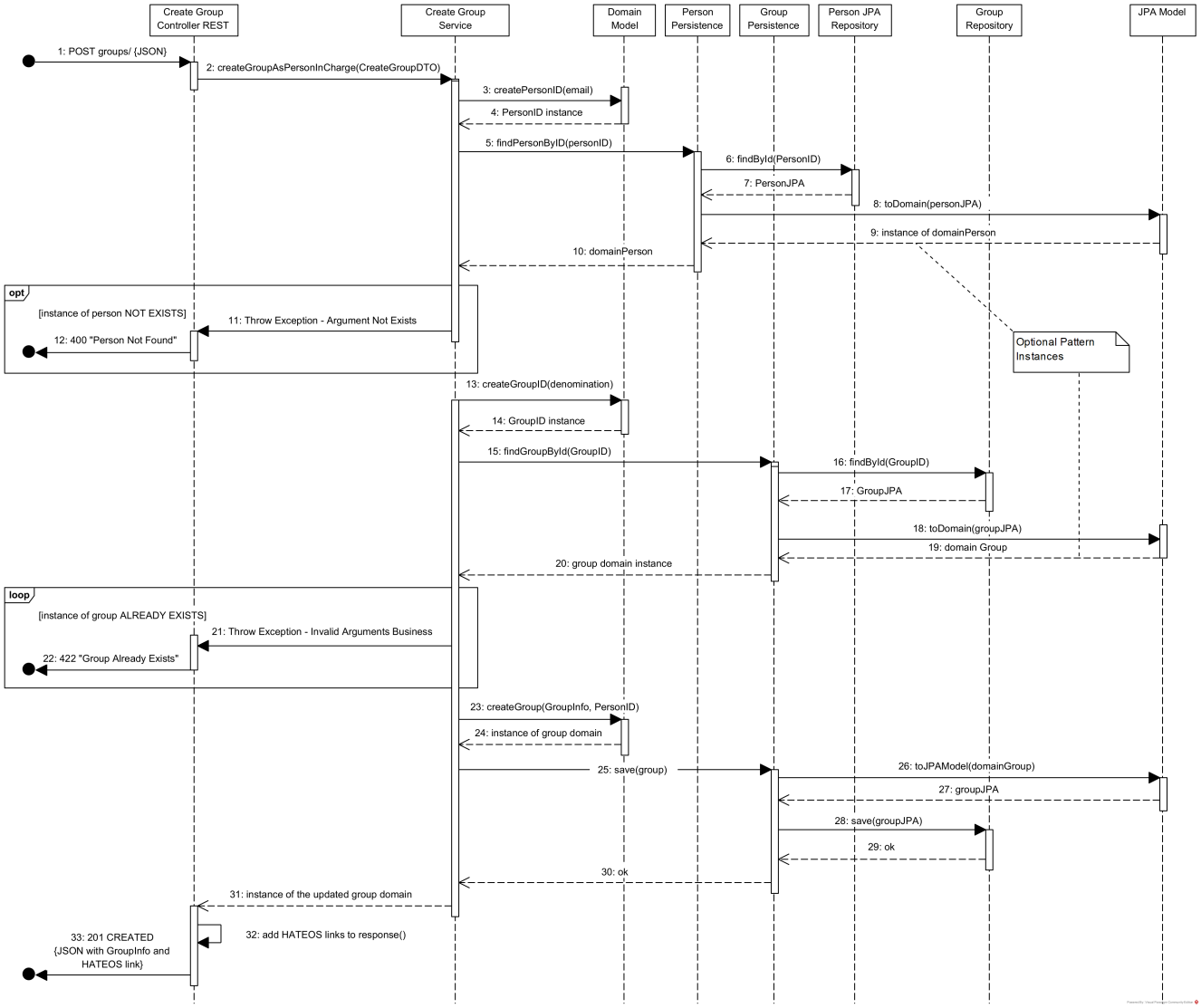

Figure 3- Sequence Diagram - Create Group as Person in charge

2. Maintainability

Use of Sonargraph

Sonargraph is a comprehensive software architecture and quality management tool designed to help maintain high-quality, maintainable, and scalable codebases. As a robust platform, Sonargraph offers a range of features that contribute to measure code quality, early detect issues, and enhance architectural understanding.

In this case, Sonargraph was used to get the Cumulative Component Dependency of the whole project, and the LoC of the classes being evaluated.

2.1. Depends Upon metric

Understanding dependencies in software development, as indicated by the "Depends Upon" metric, is crucial for maintaining code quality. This metric reveals how components rely on each other.

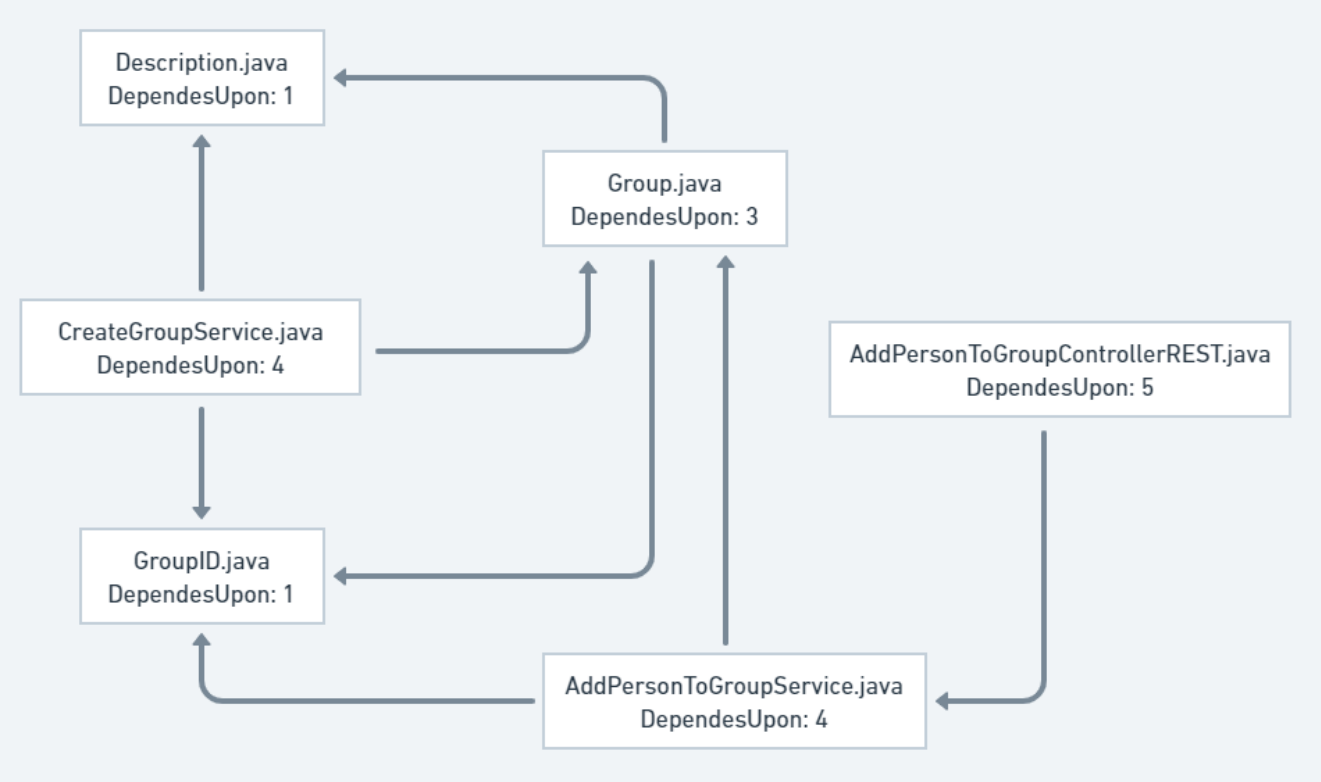

Figure 4- Depends Upon Metric

2.2. Cumulative Component Dependency (CCD) - Manually

The Cumulative Component Dependency (CCD) metric is a valuable measure in software development that provides insights into the overall complexity and interconnectivity of the codebase. It quantifies the cumulative number of dependencies that a specific class has with other elements in the system.

> A high CCD suggests a higher level of interdependence, potentially indicating greater complexity and a higher risk of cascading changes when modifications are made to that component.

```
CCD = 1 + 1 + 3 + 4 + 4 + 5 = 18
```

2.3. Cumulative Component Dependency (CCD) - SonarGraph Result (Whole Project)

By using Sonargraph, it's easy to see that de CCD of the whole project is `CCD = 7681`.

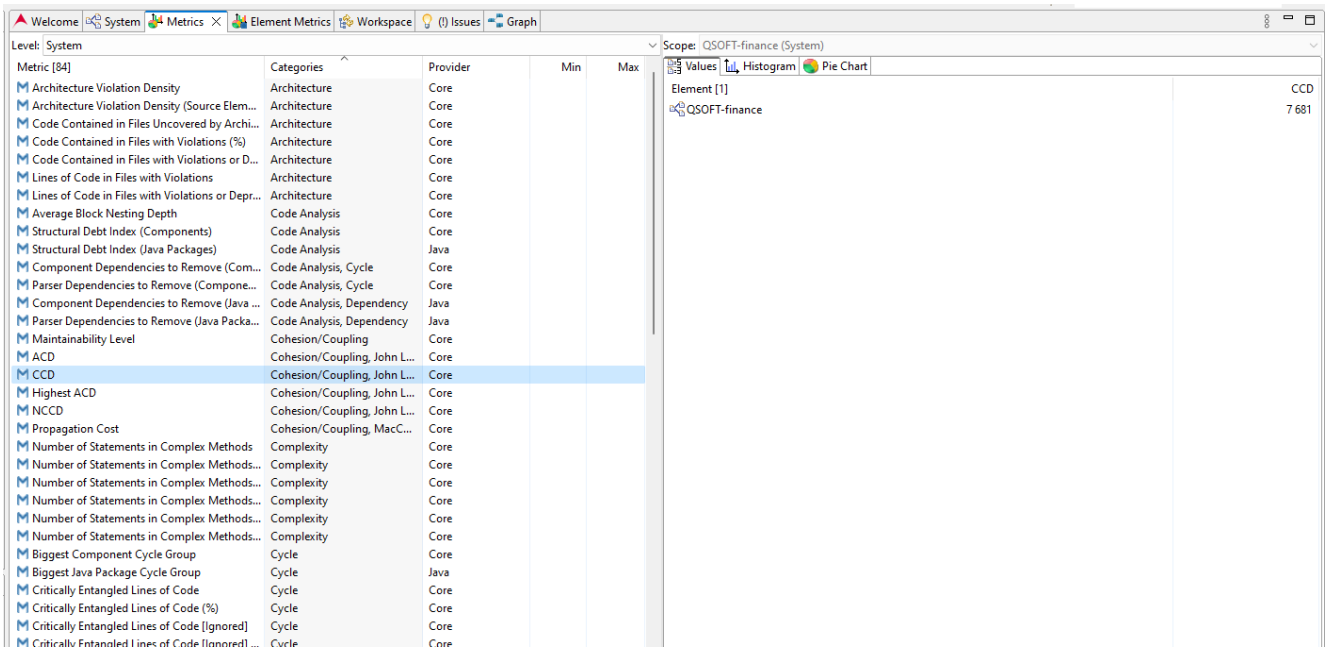

Figure 5- CCD - SonarGraph Result

2.4. Lines Of Code (LoC) per class

When it comes to defining thresholds in metrics, it usually involves establishing certain benchmarks or limits to monitor the performance of our system. These thresholds help in identifying when something is functioning as expected or when there's an anomaly or issue that needs attention. Lines of Code (LoC) per file counts every line that contains actual code and skips empty lines and comment lines. [1] Here, the thresholds were defined following the Code Quality website. From now on, the site will be called **CQ**. In the following image, it is possible to see where we can find the LoC of a class using Sonargraph. The class Group.java has 194 LoC.

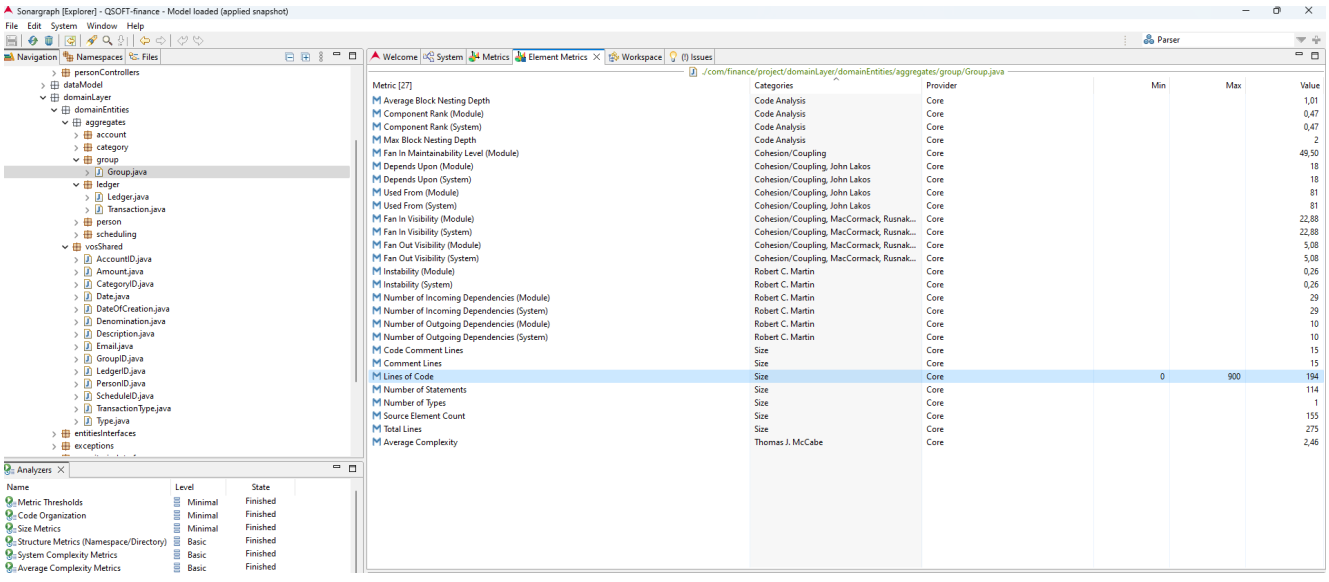

Figure 6- LoC of Group.java

As seen before, it's easy to verify the Lines of Code of every single class. With this in mind, more classes were analysed.

Class	LoC
Group.java	194
createGroupService .java	264
CreateGroupControllerREST.java	85

According to CQ, the default threshold is `1000`. So, for these classes and this metric, the application is pretty maintainable.

2.5. Number of methods per class

In software development, the structure and organization of code play a pivotal role in determining the maintainability and overall quality of the software system. One crucial aspect that directly influences these factors is the number of methods within each class.

Class	Num of methods
Group.java	27
createGroupService.java	15
CreateGroupControllerREST.java	8

According to CQ, the default threshold is `10`. So, for these classes and this metric, the application is not that interesting. The Group.java has almost 300% of the required number of methods.

GQM - What's the maintainability of production code?

While Lines of Code (LoC) per class are within acceptable thresholds, the number of methods in the Group.java and createGroupService.java class exceeds recommended limits, suggesting a potential challenge in maintainability.

So, considering the Group aggregate, the system's maintainability of production code is not acceptable.

3. Performance

Use of Apache JMeter

Apache JMeter was the tool chosen for testing the performance of the Group aggregate. This tool stands out as a robust choice for performance testing due to its combination of features.

Its versatility spans multiple protocols, including HTTP, the one we are interested in.

Altogether, JMeter emerges as a comprehensive, accessible, and powerful solution to test the performance of the Group aggregate of our XPTO system.

To run the tests (command line):

```
jmeter -n -t .\Part1\documentation\RuiNeto1230211\jmeter\jmeter.jmx -l
./Part1/documentation/RuiNeto1230211/jmeter/history
```

To generate the dashboard with results (command line):

```
jmeter -g ./Part1/documentation/RuiNeto1230211/jmeter/history/result_data.xls -o
./Part1/documentation/RuiNeto1230211/jmeter/history/{dirToSaveReport}
```

3.1. Add Person to Group

Endpoint

```
@PostMapping("/groups/{denomination}/members")
public ResponseEntity<Object> addPersonToGroupP()
```

Use of a CSV file

In conducting the performance testing analysis for the "Add Person to Group" functionality, a structured approach was utilized employing data from a CSV file. Leveraging the CSV file allowed for a systematic and reproducible testing process, as the file contains a giant list of persons' emails to add to groups.

> See Appendix 1 - CSV Generator Code.

The CSV file is located in `./jmeter/persons.csv`.

After the creation of the CSV file, it was added to the JMeter tests by using the functionality "Config File". The addition is shown below.

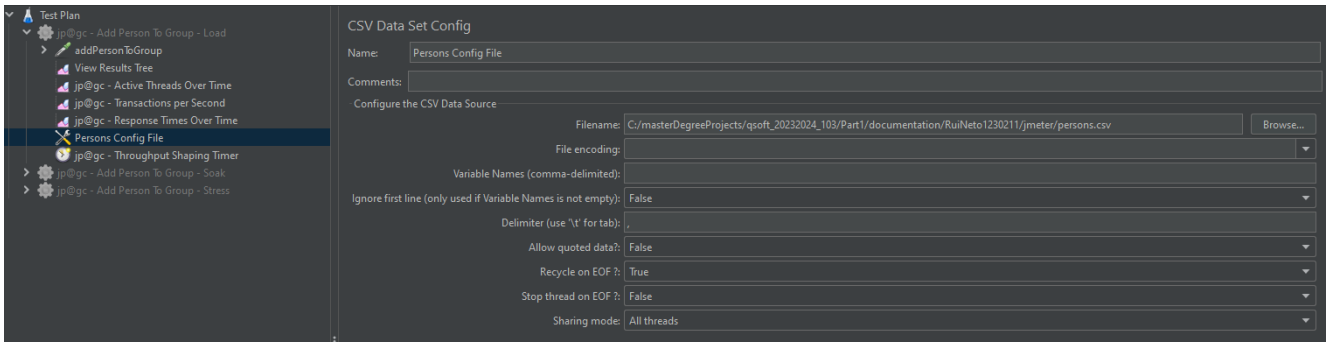

Figure 7- Persons CSV Config File JMeter

Thanks to this approach, we just needed to add `{"email":${email}}` to the request body.

3.1.1. Scenario 1

Raw Scenario

The system must process and respond to requests for adding persons to groups with an average response time of less than 2 seconds under the expected user load of 300.

Formal Scenario

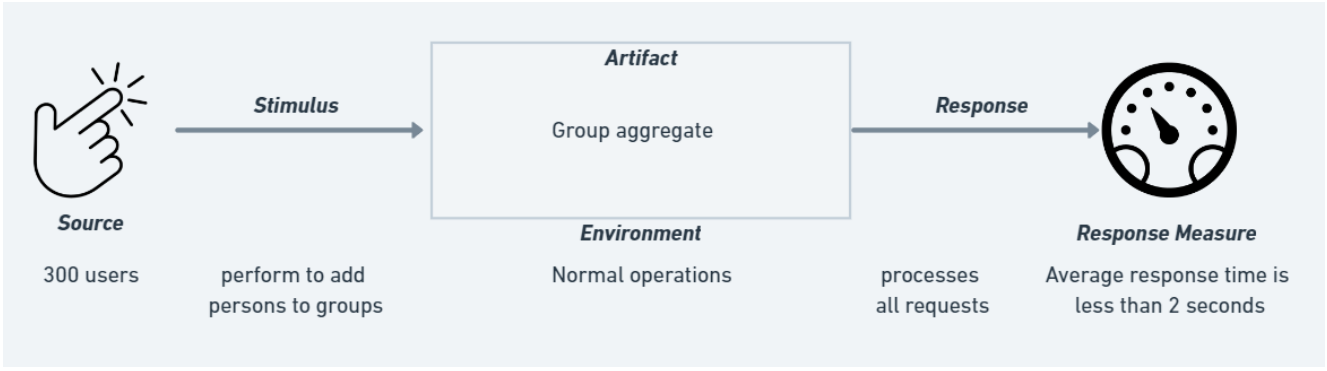

Figure 8- Formal Scenario 1

Image adaptation from [2].

Load Test

Steady Ramp-Up:

In this pattern, the number of virtual users is increased gradually and steadily over time until it reaches the desired load level. This pattern is useful for testing our system because it's possible we experience a large number of visitors during the start of the day, and maintain a more regular number during the day. Here, we should simulate a gradual increase in traffic over time, instead of a sudden spike.

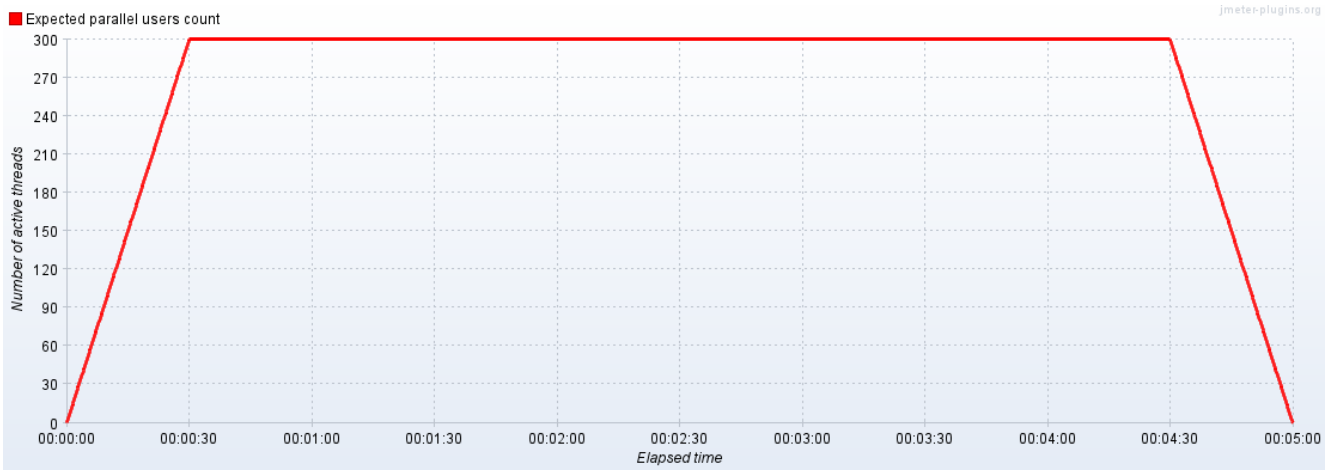

Figure 9- Load Test - Steady Ramp Up

Results:

In the image below it is possible to see that, even before the 30-second mark, the response time was already well over 2 seconds.

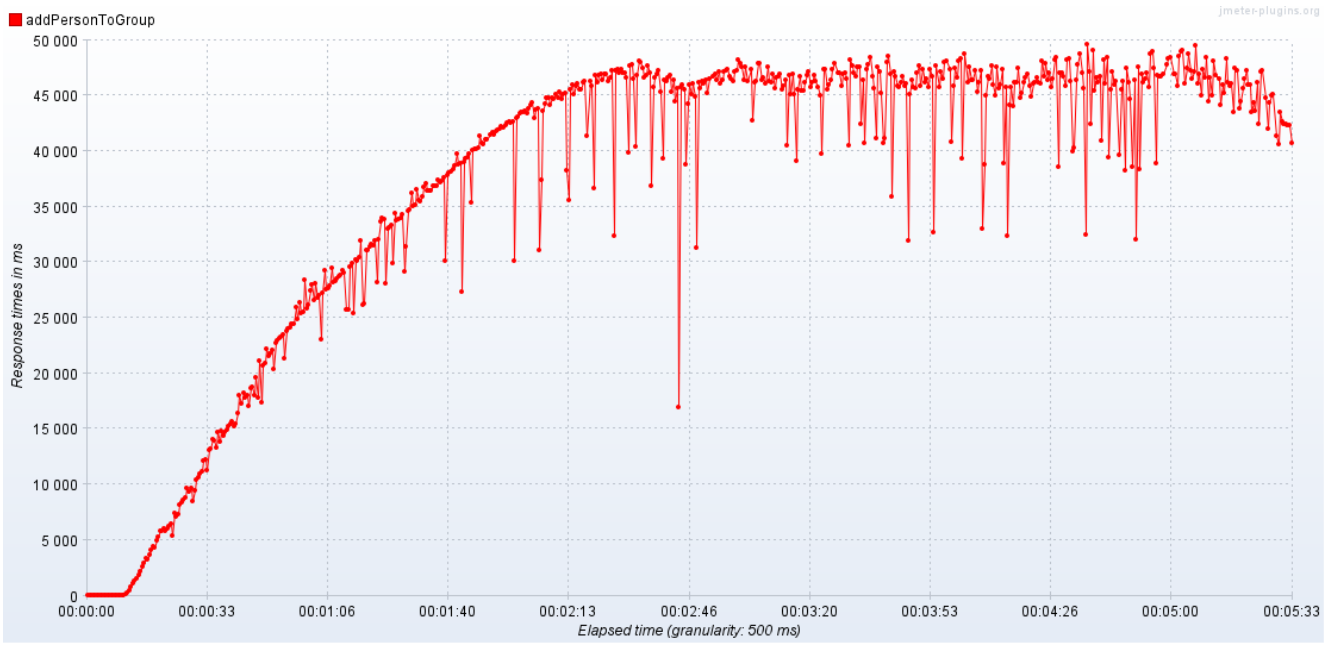

Figure 10- Load Test Result - Response Time

Now, in the image below, it's possible to see that the average response time after two minutes is over 45 seconds.

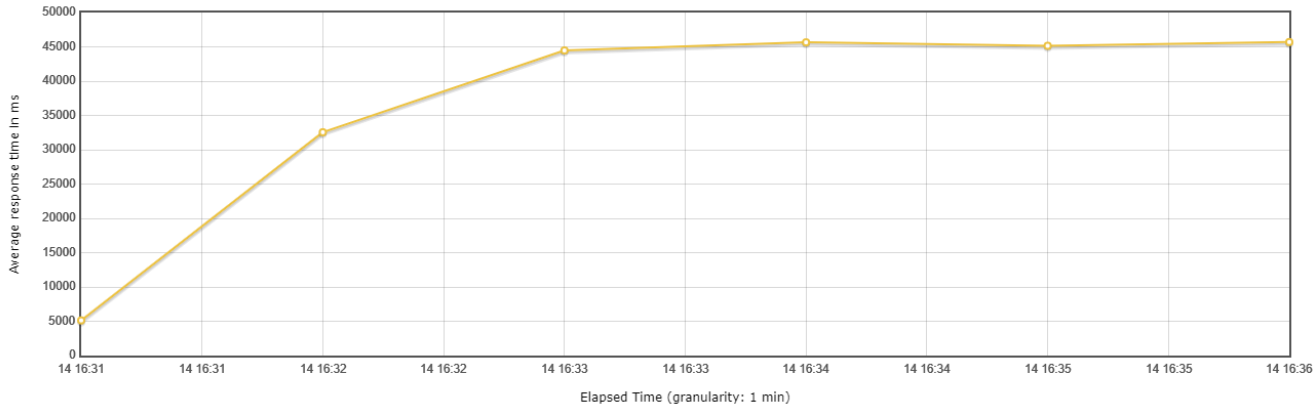

Figure 11- Load Test Result - Average Response Time

From this we find that the system needs a whole refactor in adding a person to a group. This can be critical because if these are the results from a load test, a soak test will probably be worst performed. It is documented below.

Soak test

Steady Ramp-Up:

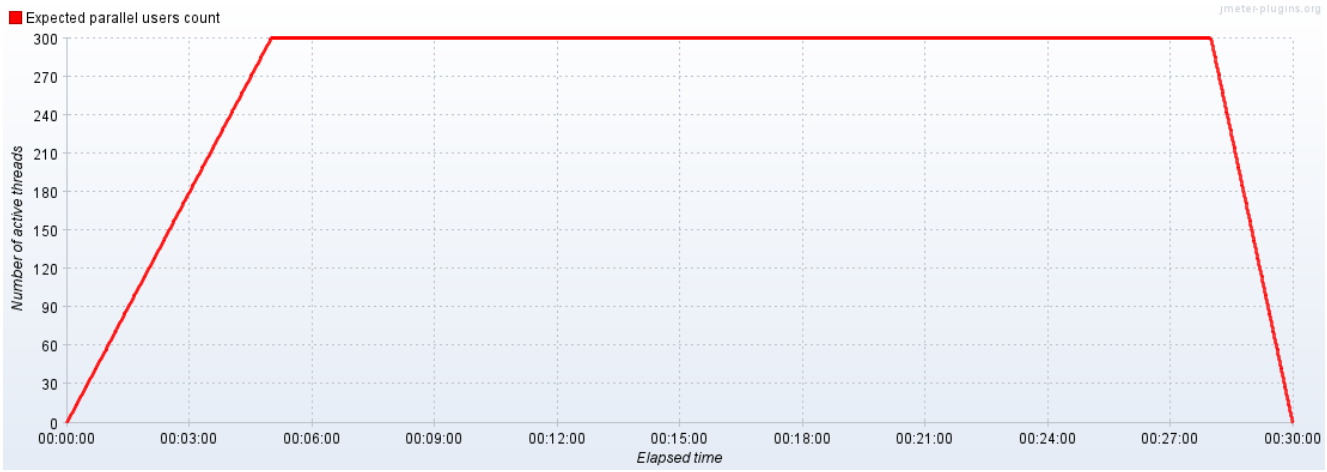

Figure 12- Soak Test - Steady Ramp Up

Results:

The following image shows that, as soon as the test starts the response time was already over the expected.

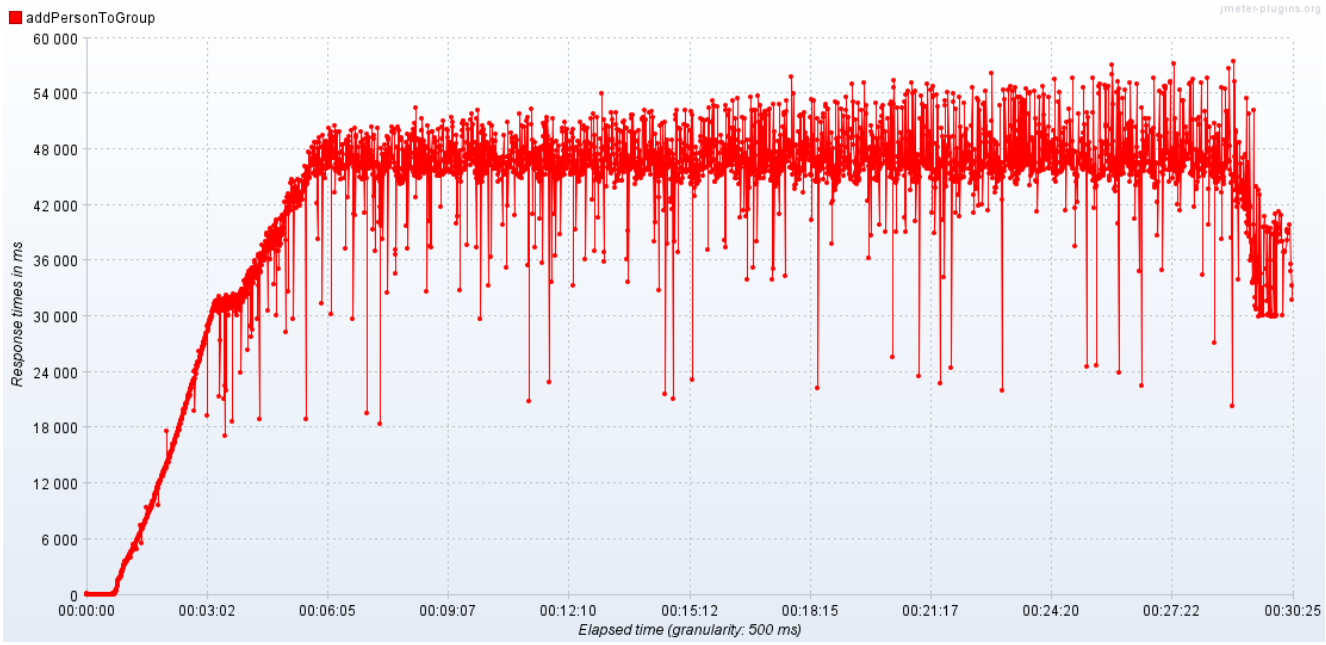

Figure 13- Soak Test Result - Response Time

Now, in the following one, it's possible to see that the average response time after 10 minutes is over 45 seconds, which is horrible for a common user to deal with in their stressful day.

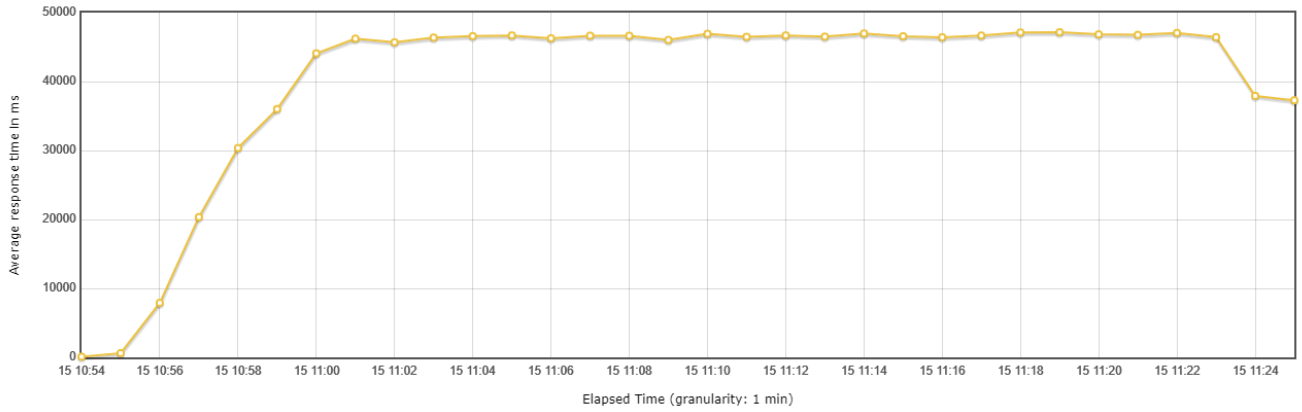

Figure 14- Soak Test Result - Average Response Time

From this view, we know that the system needs to be adapted to support a long period of requests in the context of adding a person to a group.

3.1.2. Scenario 2

Raw Scenario

The system must support the addition of persons to groups during a period of load increase equivalent to 200%, with a latency lower than 2 seconds.

Formal Scenario

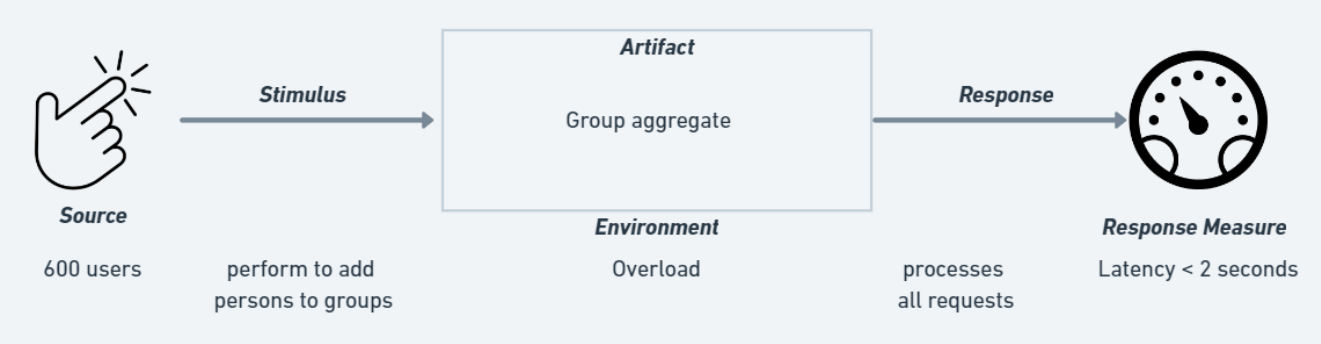

Figure 15- Formal Scenario 2

Stress test

Steady Ramp-Up:

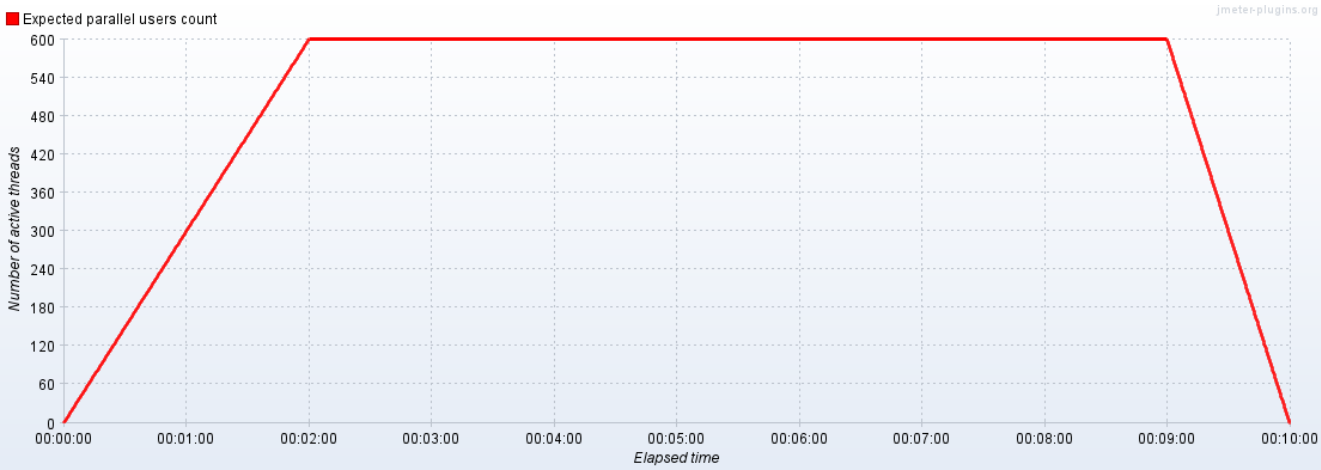

Figure 16- Stress Test - Steady Ramp Up

Results:

In the image below it is possible to see that the system is not prepared for an overload of users. At 30 seconds, the latency was already over 10 seconds. This is 5 times more than the acceptable for an operation like this.

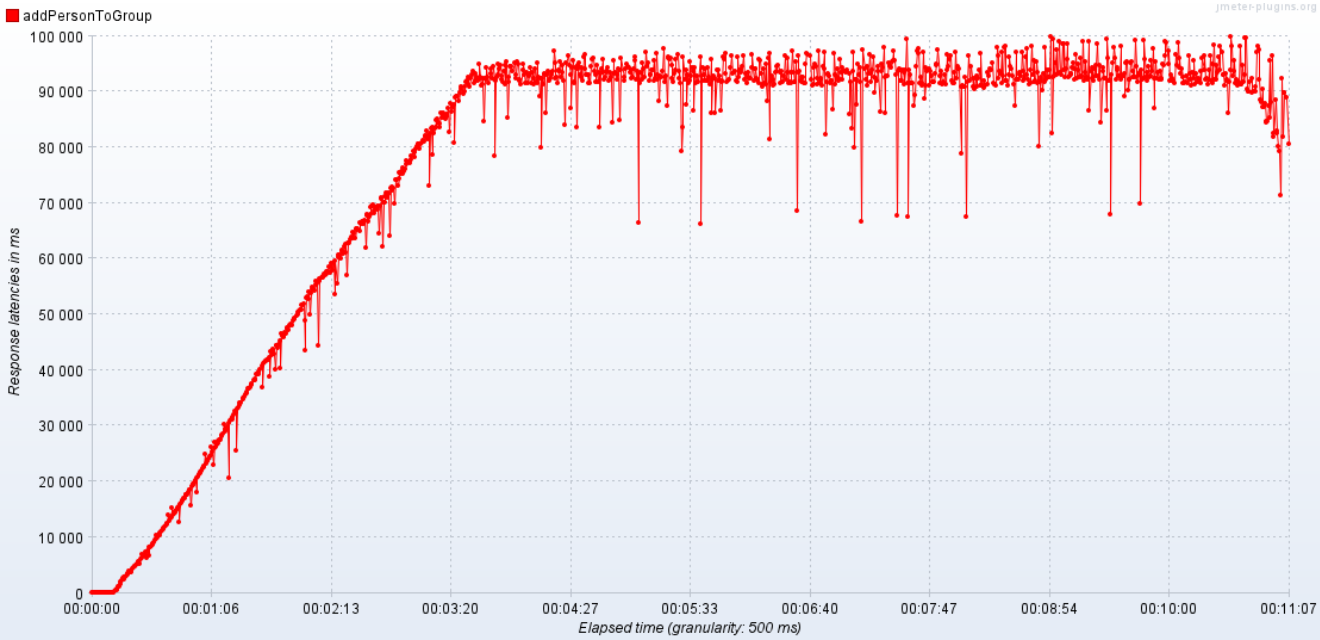

Figure 17- Stress Test Result - Latency

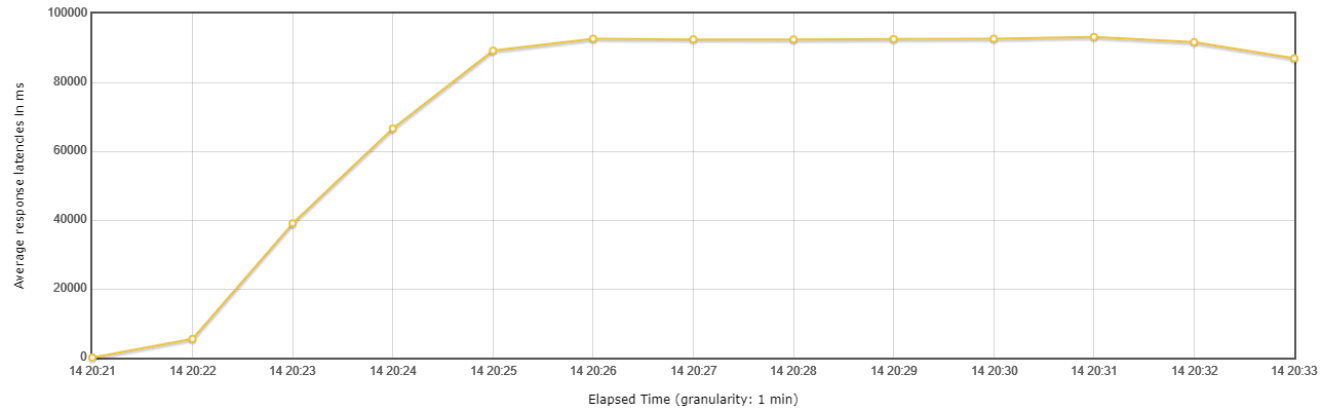

Figure 18- Stress Test Result - Average Latency

Here, we find that the system is not ready for an overload of users. In this case we doubled them, which is a very real scenario, for example, in the popular events where people spend more money and must use the system more.

3.1.3. Conclusion

The analysis of the "Add Person to Group" functionality reveals critical shortcomings in the system's performance under various scenarios. The load test indicates that even under a steady ramp-up of users, the response time exceeded the acceptable threshold of 2 seconds within the initial 30 seconds, and after 2 minutes, the average response time skyrocketed to over 45 seconds.

The soak test, designed to simulate a sustained period of activity, exposed alarming results. Within the first moments, the response time surpassed expectations, and after 10 minutes, the average response time reached an unacceptable 45 seconds mark. This highlights the urgent need for a comprehensive refactor to enhance the system's capability to handle a prolonged influx of requests during the process of adding a person to a group.

In the stress test scenario, where a 200% increase in load was simulated, the system demonstrated a notable struggle - it is not ready for an overload of users. This underscores the system's unpreparedness for handling significant spikes in user activity, a situation likely to occur during popular events or high-demand periods.

GQM - What's the system's performance?

So, considering the Group aggregate, the system's performance is not acceptable.

4. Security

4.1. Dependency vulnerability analyses

Ensuring the security of our system is paramount in today's digital landscape, where cyber threats are ever-evolving. One critical aspect of this security landscape is the thorough examination of dependencies within the system stack. In this context, the OWASP Dependency-Check tool emerges as a valuable asset, offering a systematic approach to identify and mitigate potential vulnerabilities in dependencies.

This point focuses on the use of the OWASP Dependency-Check tool in scrutinizing the `pkg:maven/com.h2database/h2@1.4.200` dependency.

To run the dependency checker, we used the command `mvn org.owasp:dependency-check-maven:check`.

Specifically into the evaluation of vulnerabilities associated with this version of the H2 database through the lens of the OWASP Dependency-Check, this examination becomes instrumental in fortifying our software against potential security risks.

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count	Confidence	Evidence Count
h2-1.4.200.jar	cpe:2.3:a:h2database:h2:1.4.200:*:*:*:*:*:*:*	pkg:maven/com.h2database/h2@1.4.200	CRITICAL	5	Highest	44

Figure 19- Dependency Analyses - pkg:maven/com.h2database/h2@1.4.200

- Highest Severity:

The severity level of the highest-rated vulnerability associated with each dependency. Severity levels typically range from low to critical, helping prioritize the resolution of security issues based on their potential impact.

- CVE Count:

This represents the total number of CVEs (Common Vulnerabilities and Exposures) associated with each dependency.

- Confidence:

The confidence level expresses the degree of certainty that the identified vulnerability is present in the dependency. A higher confidence level indicates a greater degree of certainty.

- Evidence Count:

Indicates the number of pieces of evidence or indicators that led to the identification of vulnerabilities in a particular dependency.

The **National Vulnerability Database (NVD)** provides a comprehensive overview of vulnerabilities, and in this case, we are examining the vulnerabilities associated with the H2 database version 1.4.200.

Taking a look at the NVD, it's possible to see that this dependency has a lot of critical aspects.

The vulnerabilities listed in the NVD for this specific version of the H2 database highlight the following potential security concerns that may affect systems relying on this database version.

Vuln ID	Summary	CVSS Severity
CVE-2022-45868	The web-based admin console in H2 Database Engine through 2.1.214 can be started via the CLI with the argument -webAdminPassword, which allows the user to specify the password in cleartext for the web admin console. Consequently, a local user (or an attacker that has obtained local access through some means) would be able to discover the password by listing processes and their arguments. NOTE: the vendor states "This is not a vulnerability of H2 Console ... Passwords should never be passed on the command line and every qualified DBA or system administrator is expected to know that." **Published:** novembro 23, 2022; 4:15:11 da tarde -0500	*V3.1:* 7.8 HIGH *V2.0:*(not available)
CVE-2022-23221	H2 Console before 2.1.210 allows remote attackers to execute arbitrary code via a jdbc:h2:mem JDBC URL containing the IGNORE_UNKNOWN_SETTINGS=TRUE;FORBID_CREATION=FALSE;INIT=RUNSCRIPT substring, a different vulnerability than CVE-2021-42392. **Published:** janeiro 19, 2022; 12:15:09 da tarde -0500	*V3.1:* 9.8 CRITICAL *V2.0:* 10.0 HIGH
CVE-2021-42392	The org.h2.util.JdbcUtils.getConnection method of the H2 database takes as parameters the class name of the driver and URL of the database. An attacker may pass a JNDI driver name and a URL leading to a LDAP or RMI servers, causing remote code execution. This can be exploited through various attack vectors, most notably through the H2 Console which leads to unauthenticated remote code execution. **Published:** janeiro 10, 2022; 9:10:23 da manhã -0500	*V3.1:* 9.8 CRITICAL *V2.0:* 10.0 HIGH
CVE-2021-23463	The package com.h2database:h2 from 1.4.198 and before 2.0.202 are vulnerable to XML External Entity (XXE) Injection via the org.h2.jdbc.JdbcSQLXML class object, when it receives parsed string data from org.h2.jdbc.JdbcResultSet.getSQLXML() method. If it executes the getSource() method when the parameter is DOMSource.class it will trigger the vulnerability. **Published:** dezembro 10, 2021; 3:15:07 da tarde -0500	*V3.1:* 9.1 CRITICAL *V2.0:* 6.4 MEDIUM

Figure 20- Dependency Analyses - vulnerabilities

With this mind, it's clear that the system needs an urgent upgrade to a more recent version to ensure the benefits from the latest security patches and enhancements.

Looking at the MVN Repository, it's easy to find that these versions have no vulnerabilities:

- 2.2.224
- 2.2.222
- 2.2.220

4.2. Class analyses

The following table has been defined with the aim of making the evaluation of the data inputs of the classes more coherent.

Measurement	Rational
1	No validations
2	Has validations only in terms of technical perspective*
3	Has validations only in terms of technical and domain perspective**

> *Technical perspective: The software checks for validations in terms of java compilation errors, e.g., check if the instance is not null, the list doesn't contain null values, etc., but ignores the domain validation, an email's String has the same treatment as an address' String.
>
> **Domain perspective: In addition to technical validations, the software checks for validations in terms of the domain, e.g, if the email is on the expected format.

4.2.1. Group.java

The Group.java class incorporates technical validations by checking for null values in critical parameters during the creation of a Group instance. For instance, it verifies that the personInCharge parameter and the members are not null. However, there is room for improvement as it does not verify if the description, for example, is null.

So, the class falls under 1. It's worth noting that the class has other constructive elements, such as encapsulation through private fields and methods, proper use of constructors, and adherence to good practices like creating immutable objects.

```
public class Group implements Entity, Owner {
    private Group(String denomination, List<PersonID> peopleInCharge,
List<PersonID> members,
                  String description, LocalDate dateOfCreation, LedgerID ledgerID)
{

        if (peopleInCharge == null) {
            throw new NullPointerException("Group not created. People in charge
can't be Null");
        } else {
            this.peopleInCharge = peopleInCharge;
        }

        if (members == null) {
            throw new NullPointerException("Group not created. Members can't be
Null");
        } else {
            this.members = members;
        }

        this.description = Description.createDescription(description);
        this.dateOfCreation = DateOfCreation.createDateOfCreation(dateOfCreation);
        this.groupID = GroupID.createGroupID(denomination);
        this.ledgerID = ledgerID;
    }
}
```

4.2.2. GroupID.java

The GroupID.java class performs technical validations during the creation of a GroupID instance. It checks if the denomination parameter is null or an empty string, and throws an IllegalArgumentException if the condition is met.

This ensures that a GroupID is created with a valid Denomination. This aligns with both criteria of having technical and domain validations.

```
public class GroupID implements OwnerID, ValueObject, Serializable {
    private GroupID(String denomination) {

        if (denomination == null || denomination.equals("")) {
            throw new IllegalArgumentException("GroupID not created. The
denomination parameter " +
                    "can't be null or a empty string");
        } else {
            this.denomination = Denomination.createDenomination(denomination);
        }
    }
}
```

4.2.3. DateOfCreation.java

The constructor performs a technical validation by checking if the dateOfCreation parameter is null. If it is null, an IllegalArgumentException is thrown. This ensures that a valid LocalDate is provided during the creation of a DateOfCreation instance.

But it doesn't verify if the data is empty, so it's possible to verify that the class doesn't respond to the domain perspective, where a valid date should be mandatory.

```
public class DateOfCreation implements ValueObject, Serializable {
    private DateOfCreation(LocalDate dateOfCreation) {

        if (dateOfCreation == null) {
            throw new IllegalArgumentException("DateOfCreation not created due to
the fact that the " +
                    "dateOfCreation parameter hasn't a valid argument");
        }

        this.dateOfCreation = dateOfCreation;
    }
}
```

4.2.4. All Results

The following table shows the result of the 7 classes chosen to evaluate.

Class	Evaluation
Group.java	1
LedgerID.java	1
PersonID.java	2
DateOfCreation.java	2
AccountID.java	2
GroupID.java	3
ScheduleID.java	3

GQM - What are the system's security problems?

Pointing to the security analysis using the OWASP Dependency-Check tool for the H2 database and the examined classes, both has revealed critical vulnerabilities, which is not good point for the reuse of the system.

So, considering the Group aggregate, the system's security problems are not acceptable.

5. Architectural compliance

Architectural compliance is a critical aspect of software development and maintenance, ensuring that the implemented architecture aligns with the intended design principles and organizational standards. This process involves continuously monitoring and validating the codebase and system against the established architectural specifications.

Use of ArchUnit

ArchUnit is a simple but powerful open-source library to automatically test Java architectures as plain unit tests. [3] As a unit testing framework, ArchUnit enabled to define and execute tests that check **package and class** dependencies.

5.1. Package Dependency Check

Fitness Function

- **Breadth of feedback:** Holistic (will target the whole system)
- **Execution trigger:** Periodical and in the test environment
- **Metric type:** Binary
  - True: The system complies with the conventions about Package Dependency.
  - False: The system doesn't comply with the conventions about Package Dependency.
- **Automated:** No
- **Quality attribute requirements:** Maintainability
- **Static or dynamic:** Static

The following image is useful to understand how the tests works. In the required case, we tested if the `groupControllers`, `personControllers` and `otherControllers` packages depend on classes that resides in the `appllicationServices` or `dtos` packages.

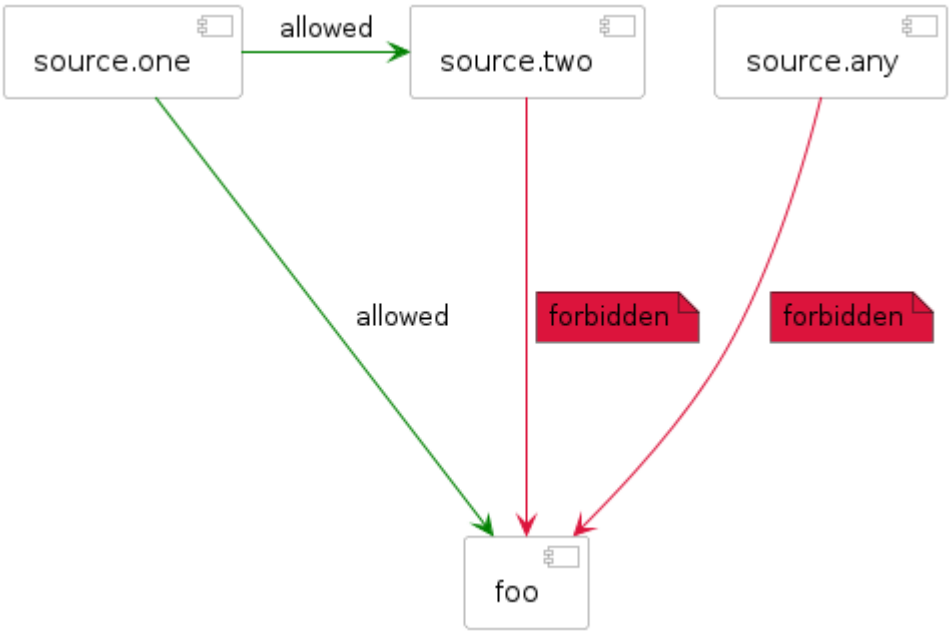

Figure 21- Package Dependency Checks Example

Test code

```
public class PackageDependencyCheckTest {
    @ArchTest
    static void checkGroupControllersPackageDependencyTest(JavaClasses classes) {
        classes().that().resideInAPackage("..groupControllers..")

.should().dependOnClassesThat().resideInAnyPackage("..applicationServices..",
"..dtos..")
                .check(classes);
    }

    @ArchTest
    static void checkPersonControllersPackageDependencyTest(JavaClasses classes) {
        classes().that().resideInAPackage("..personControllers..")

.should().dependOnClassesThat().resideInAnyPackage("..applicationServices..",
"..dtos..")
                .check(classes);
    }

    @ArchTest
    static void checkOtherControllersPackageDependencyTest(JavaClasses classes) {
        classes().that().resideInAPackage("..otherControllers..")

.should().dependOnClassesThat().resideInAnyPackage("..applicationServices..",
"..dtos..")
                .check(classes);
    }
}
```

Results

With the tests' results, it is possible to see that the system complies with good conventions about the package dependency, keeping in mind that all the 3 tests passed.

However, this fitness function only cover the controllers packages dependencies. To ensure that the whole system follows good conventions, more tests are needed.

5.2. Class Dependency Check

Fitness Function

- **Breadth of feedback:** Holistic (will target the whole system)
- **Execution trigger:** Periodical and in the test environment
- **Metric type:** Binary
  - True: The system complies with the conventions about Class Dependency.
  - False: The system doesn't comply with the conventions about Class Dependency.
- **Automated:** No
- **Quality attribute requirements:** Maintainability
- **Static or dynamic:** Static

The following image is useful to understand how the tests works. In the required case, we tested if the ended with `ControllerREST` classes only have dependent classes ended with `Service` or `DTO`.

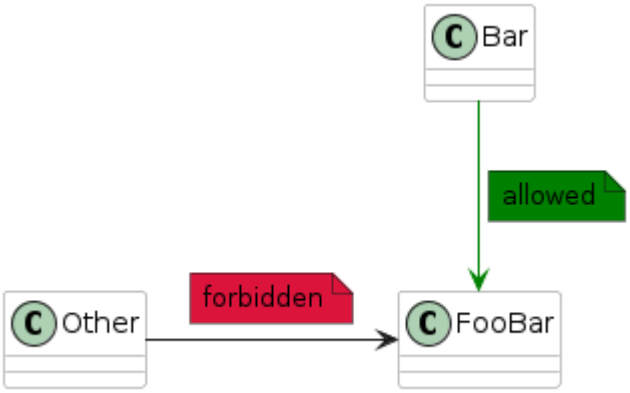

Figure 22- Class Dependency Checks Example

Test code

```
public class ClassDependencyCheckTest {
  @ArchTest
  static void checkControllersClassDependencyTest(JavaClasses classes) {
    classes().that().haveNameMatching(".*ControllerREST")
            .should().onlyHaveDependentClassesThat()
            .haveSimpleName(".*Service")
            .orShould().haveSimpleName(".*DTO")
            .check(classes);
  }
}
```

GQM - Does the system consistently follow architectural principles?

With the tests' results, it is possible to see that the system doesn't comply with good conventions about the class dependency, keeping in mind that this simple test failed. So, considering the Group aggregate, the system's architectural principles are not acceptable.

6. Maintainability of test code

The term code smell indicates symptoms that may indicate deeper problems in the system's source code. Several studies show that code smells hinder the comprehensibility and maintainability of software systems. They increase the risk of bugs or failures in the future

And test code? Software testing is an essential part of the software development life cycle. Testing code matters.

6.1. Test Smells

Test smells are suboptimal design choices developers make when implementing test cases [4]. They are prevalent in real life and damage the maintenance and comprehensibility of the test suite [5].

The following image outlines the test smells found in certain classes within the Group aggregate.

6.1.1. GroupTest.java

LOC	numberMethods	Assertion Roulette	Eager Test	Mystery Guest	Sleepy Test	Unknown Test	Redundant Assertion	Dependent Test	Magic Number Test	Conditional Test Logic	EmptyTest	General Fixture	IgnoredTest	Sensitive Equality	Verbose Test	Default Test	Resource Optimism	Duplicate Assert	Exception Catching Throwing	Constructor Initialization	Print Statement	Lazy Test
2357	40	49	35	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	3	0	0	22

Figure 23- Test smells - GroupTest.java

Tests smells found:

- 49 Assertion Roulette
- 35 Eager Test
- 2 Verbose Test
- 3 Exception Catching Throwing
- 22 Lazy Test

Examples:

- *Assertion Roulette*

Occurs when a test method has multiple non-documented assertions. Multiple assertion statements in a test method without a descriptive message impact readability/understandability/maintainability as it's not possible to understand the reason for the failure of the test. [6]

Detection: A test method contains more than one assertion statement without an explanation/message (parameter in the assertion method). [6]

```
public class GroupTest {
    @Test
    @DisplayName("Verify if one CategoryID can be added to Categories List || Sad
case: CategoryID already exist")
    public void categoryIDNotAddedToCategoriesList() {
        //...
        assertEquals(true, result1);
        assertEquals(false, result2);
        assertEquals(groupFriends.getCategories(), categories);
    }
}
```

Here, The assertEquals() method is called 3 times. Each assert statement checks for a different condition, without an explanation message for each assert statement. If one of the assert statements fails, identifying the cause of the failure is not simple.

- *Eager Test*

Tests are often eager as they test (entirely) unrelated functionalities. The checking of an object's state using multiple getter calls after some action is rarely avoidable. [7]

If the test is cohesive enough and focuses on a single feature, the assertions should ensure that the entire behavior is as expected. This may mean asserting that many fields were updated and have a new value. [7]

```
public class GroupTest {
    @Test
    @DisplayName("Verify if one CategoryID can be added to Categories List || Sad
case: CategoryID already exist")
    public void categoryIDNotAddedToCategoriesList() {
        //...
        assertEquals(true, result1);
        assertEquals(false, result2);
        assertEquals(groupFriends.getCategories(), categories);
    }
}
```

This test is verifying, at the same time, if the category is successfully added and then gets all the categories. Tests should not have unrelated assertions.

- *Verbose Test*

This smell occurs when the tests use a lot of code to do what they need to do. In other words, the test code is not clean and simple. [8]

```
public class GroupTest {
    @Test
    @DisplayName("Get records of a account in a determined period of time (only
one movement are valid due to the use of a different account)| Happy Case
(different account)")
    public void testGetRecordsBetweenTwoDatesOfADeterminedAccountDebitAndCredit()
{
        //100 lines of code
    }
}
```

This test has almost 100 lines of code because it's adding 4 transactions when it should do the same job with only 2. This makes the test maintainability harder.

6.1.2. AddPersonToGroupControllerRESTTest.java

LOC	numberMethods	Assertion Roulette	Eager Test	Mystery Guest	Sleepy Test	Unknown Test	Redundant Assertion	Dependent Test	Magic Number Test	Conditional Test Logic	EmptyTest	General Fixture	IgnoredTest	Sensitive Equality	Verbose Test	Default Test	Resource Optimism	Duplicate Assert	Exception Catching Throwing	Constructor Initialization	Print Statement	Lazy Test
151	4	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 24- Test smells - AddPersonToGroupControllerRESTTest.java

Tests smells found:

- 1 Assertion Roulette
- 1 UnknownTest

6.1.3. CreateGroupServiceTest.java

LOC	numberMethods	Assertion Roulette	Eager Test	Mystery Guest	Sleepy Test	Unknown Test	Redundant Assertion	Dependent Test	Magic Number Test	Conditional Test Logic	EmptyTest	General Fixture	IgnoredTest	Sensitive Equality	Verbose Test	Default Test	Resource Optimism	Duplicate Assert	Exception Catching Throwing	Constructor Initialization	Print Statement	Lazy Test
747	30	15	15	0	0	0	0	0	0	0	0	0	0	0	8	0	0	0	0	0	0	8

Figure 25- Test smells - CreateGroupServiceTest.java

Tests smells found:

- 15 Assertion Roulette
- 15 EagerTest
- 8 VerboseTest
- 8 LazyTest

GQM - What's the maintainability of test code?

The analysis of test code maintainability reveals the presence of various test smells in the test suites for the Group aggregate, such as Assertion Roulette, Eager Test, Verbose Test, and Lazy Test have been identified in specific test classes.

So, considering the Group aggregate, the system's test code maintainability is not acceptable.

7. Conclusions

Talking about maintainability, while Lines of Code (LoC) per class are within acceptable thresholds, the number of methods in the Group.java class exceeds recommended limits, suggesting a potential challenge in maintainability. However, the application performed well in the other maintainability tests.

The performance testing of the "Add Person to Group" functionality revealed critical deficiencies in the system's performance. The load, soak and stress tests showed that response times and latencies exceeded acceptable thresholds. These findings underscore for a comprehensive opinion that the system cannot be reused.

Now, pointing to the security analysis using the OWASP Dependency-Check tool for the H2 database and the examined classes, both has revealed critical vulnerabilities, which is not good point for the reuse of the application.

The Package Dependency Check tests indicate that the system currently complies with good conventions. On the other hand, the Class Dependency Check tests highlight a deviation from conventions.

The analysis of test code maintainability reveals the presence of various test smells in the test suites for the Group aggregate, such as Assertion Roulette, Eager Test, Verbose Test, and Lazy Test have been identified in specific test classes.

After carefully weighing all the observed aspects, it is evident that the application has a considerable number of issues that require substantial time for resolution. This makes it an unsuitable reference for reuse in a different context.

So all the quality attributes tested in the Group aggregate are not acceptable.

References

[1] *Christian Ciceri, Dave Farley, Neal Ford, Andrew Harmel-Law, Michael Keeling, Carola Lilienthal, João Rosa, Alexander von Zitzewitz, Rene Weiss, and Eoin Woods. 2022. Software Architecture Metrics (1st ed.). O'Reilly Media*

[2] *L. Bass, P. Clements, and R. Kazman. 2021. Software Architecture in Practice (4 ed.). Addison-Wesley Professional.*

[3] *Peter Gafert. [n.d.]. About ArchUnit. https://www.archunit.org/about*

[4] *Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D Newman, Abdullatif Ghallab, and Stephanie Ludi. 2021. Test smell detection tools: A systematic mapping study. Evaluation and Assessment in Software Engineering (2021), 170–180.*

[5] *Maurício Aniche. 2022. Effective Software Testing: A developer's guide. Manning Publications Co*

[6] *Test Smell. [n.d.]. Software Unit Test Smells. https://testsmells.org/*

[7] *Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. 2020. Revisiting test smells in automatically generated tests: limitations, pitfalls, and opportunities. In 2020 IEEE international conference on software maintenance and evolution (ICSME). IEEE, 523–533.*

[8] *Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001). Citeseer, 92–95*

Appendices

Appendix 1 - CSV Generator Code

```
public class Bootstrapping {
    String csvFilePath =
".\\qsoft_20232024_103\\Part1\\documentation\\RuiNeto1230211\\jmeter\\persons.csv"
;
    try(CSVWriter csvWriter = new CSVWriter(new FileWriter(csvFilePath))){
        // Write the header
        String[] header = {"email", "name", "birthdate", "birthplace"};
        csvWriter.writeNext(header);
        // Generate and write data to CSV
        DateTimeFormatter testFormatter = DateTimeFormatter.ofPattern("yyyy-MM-
dd");
        for (int i = 1; i <= 200000; i++) {
            String testEmail = "test" + i + "@gmail.com";
            String testName = "Teste " + i;
            String testBirthdate = LocalDate.of(2002, 2,
20).format(testFormatter);
            String testBirthplace = "Santo Tirso";
            // Write data to CSV
            String[] data = {testEmail, testName, testBirthdate, testBirthplace};
            csvWriter.writeNext(data);
            CreatePersonDTO createPersonDTO =
CreatePersonDTOAssembler.createDTOFromPrimitiveTypes(testEmail, testName,
                    testBirthdate, testBirthplace);
            PersonID testPersonID = PersonID.createPersonID(testEmail);
            createPersonService.createAndSavePerson(createPersonDTO);
            createPersonService.addAddressToPerson(testPersonID, porto);
        }
        System.out.println("CSV file generated successfully.");
    } catch(IOException e){
        e.printStackTrace();
    }
}
```