Riccardo Grieco, Anisja Mayer
Lina Mehrle, Dominique Ostermayer

Detailed Quality Assurance Conecept

This document provides insights how we plan to ensure quality of the software we develop. In general all team members are responsible for assuring the quality of our software and the ones coding should also be the ones testing their code snippets. The overall QA is Anisja's and Riccardo's responsibility. They will peer-review the code of every team member, will check the unit tests if written and throw and catch additional exceptions for a more fine-grained error localization, if needed.

1 Constructive Quality Management

1.1 Technical Procedure

1.1.1 Follow the guidelines

Our software's guide is the requirements file authored by Lina for milestone 1. It encompasses crucial rules governing the game logic, GUI, and network protocol. This document serves as the cornerstone of our software development, providing a comprehensive reference for all code. As we progress with the project and delve into finer details, each team member is responsible for updating this file. All decisions that need to be done are written down in this file.

1.1.2 Documentation

We place a strong emphasis on thorough documentation throughout our codebase to ensure clarity and ease of understanding for all team members. Our approach involves using Javadoc comments to provide concise summaries of the main purpose and functionality of each method. These comments serve as a quick reference point for developers seeking to understand the role of a particular method within the code. In addition to Javadoc comments, we employ single-line or multi-line comments to offer detailed explanations of the implementation logic. These comments dive deeper into the intricacies of the code, elucidating complex thoughts and methodologies for developers who require a more in-depth understanding. Furthermore, we are committed to adhering to established conventions and best practices in our documentation efforts. Consistency in our documentation style ensures that team members can navigate the codebase with ease, promoting collaboration and efficiency in our development process. By maintaining meticulous documentation, we not only foster a deeper understanding of the codebase among our team members but also pave the way for seamless future maintenance and expansion of our software.

1.1.3 Coding Conventions

We adhere to coding conventions to maintain cohesive, visually appealing, and easily reviewable code. Following these conventions ensures that our code is not only aesthetically pleasing but also facilitates smoother code reviews. We stick to the intelliJ default coding and formatting conventions.

1.1.4 Meaningful Exception Handling

We find ourselves navigating exception handling much like a game of catch – tossing them where necessary and ensuring we catch them as well. It's a fundamental aspect of our coding practice, allowing us to address expected issues that can occur promptly and effectively. Whenever exceptions occur, we take care to handle them thoughtfully, considering the specific context in which they arise. This approach ensures that our code remains robust and resilient, even in the face of unexpected errors.

1.1.5 UNIT Tests

To assure the functions and quality of the game, especially the networking and the game logic, we test every class with unit tests. Here is our unit test conventions list:

1) Only important/Fundamental methods get tested.
2) Exceptions always get tested. Those tests are named *classname* + "ExceptionsTest".
3) The test class is named like the original class + "Test", for example: `Dice.java` has the test class `DiceTest.java`.
4) The method that gets tested is called like the original method + "Test", for example: `fullHouse()` is the original method and `fullHouseTest()` is the test method.
5) The correct structure and naming of the test files is ensured with the *CRTL* + *Shift* + *t* combination, that gets pressed in the class we want to get a test class for.
6) No *happy testing* and ensure random unit tests.
7) We use lambda expressions in tests so they run through and do not stop, when first test fails

Not all methods in every class will be tested, only the core critical ones like the core components. Methods that only return a value (in a different format), like the following one:

```
public String getName() { return name; }
```

will not be tested. Only complex methods that are fundamental for the networking, game and its mechanics will be tested with unit tests, just like the following code snippet that ensures that an integer array (set of combinations) gets tested for a full house combination:

```
public static int fullHouse(int[] rolledDice) throws Exception {
    if (!(rolledDice.length == 5)) {
        throw new Exception("There are 5 dice, but you handed me more or less.");
    }

    int res = 0;
    Arrays.sort(rolledDice);
    boolean tripletFirstPairLast = rolledDice[0] == rolledDice[2]
                                && rolledDice[3] == rolledDice[4];
    boolean pairFirstTripletLast = rolledDice[0] == rolledDice[1]
                                && rolledDice[2] == rolledDice[4];
    if (tripletFirstPairLast || pairFirstTripletLast) {
        res = 25;
    }
    return res;
}
```

which is an important method for the game and has a higher complexity. We will skip happy testing, but we will sometimes need to test specific cases, like seen in `GameManagerTest.java` class.
For example if we want to test our `fullHouse()` method from earlier, we need to hand specific combinations to the method to test, if it really recognises a full house (a pair and a triplet). We then test combinations that are a full house and some that are not. A meaningful unit test could look like the following:

```
int[] threes = {3, 3, 3, 3, 3};
int[] fives = {5, 1, 5, 5, 5};
int[] fullHouse = {6, 4, 6, 4, 6};

@Test
@DisplayName("Checks if full house gets detected.")
void fullHouseTest(){
    assertAll(() -> assertEquals(0, GameManager.fullHouse(fives)),
        () -> assertEquals(25, GameManager.fullHouse(fullHouse)),
        () -> assertEquals(25, GameManager.fullHouse(threes))
    );
}
```

where we test different combinations given by the arrays above. Only with specific combinations we can assure that the method can identify a full house correctly. For other methods like the following `rollDice()` method:

```
public boolean rollDice() {
    boolean couldRoll = false;
    if (!(savingStatus) && numberOfRolls < 3) {
        numberOfRolls = numberOfRolls + 1;
        diceValue = (int) Math.floor(Math.random() * 6 + 1);
        if (numberOfRolls == 3) {
            saveDice();
        }
        couldRoll = true;
    }
    return couldRoll;
}
```

we use randomly generated values and check, if this method returns us values between 1 and 6. You can see an example of such a unit test in the following code:

```
void rollDiceTest(){
        Dice dice2 = new Dice();
        Dice dice3 = new Dice();
        dice2.rollDice();
        dice3.rollDice();
        dice3.rollDice();
        dice3.rollDice();

        assertAll(() -> assertTrue(dice2.getDiceValue() <= 6 && dice2.getDiceValue() >= 1),
```

```
                () -> assertTrue(dice3.getDiceValue() <= 6 && dice3.getDiceValue() >= 1)
                );
    }
```

When an exception is thrown in a method, we **always** test this with a unit test. The exception is tested with the `assertThrows()` method. For example: the `fullHouse` method from earlier throws an exception, if it gets handed an array that is of length less than 5 or greater than 5. The exception test will look like this:

```
void gameManagerExceptionsTest() {
    // generate array of random length between 6 and 100 with numbers of values
    // between 1 and 6 inside
    int[] largeRandomArray = new int[(int) Math.floor(Math.random() * 100 + 6)];
    for (int num : largeRandomArray) {
        num = (int) Math.floor(Math.random() * 6 + 1);
    }

    // generate array of random length between 1 and 5 with numbers of values
    // between 1 and 6 inside
    int[] smallRandomArray = new int[(int) Math.floor(Math.random() * 4 + 1)];
    for (int num : smallRandomArray) {
        num = (int) Math.floor(Math.random() * 6 + 1);
    }

    assertAll(
        () -> assertThrows(Exception.class, () -> GameManager.fullHouse(largeRandomArray)),
        () -> assertThrows(Exception.class, () -> GameManager.fullHouse(smallRandomArray))
    );
}
```

So we test different sizes and use the `assertThrows()` method to test if the method throws exceptions correctly.

1.2 Organisation

1.2.1 Shared Allocation of Tasks

We maintain a Google Excel sheet accessible only to our group, serving as a central hub for task assignment and tracking across milestones. Each task is assigned to a responsible individual, who updates the status as *will do*, *in progress* or *done*. This system ensures that every task is accounted for and progress is transparent to all team members. It facilitates effective workload management, allowing us to identify instances where someone may need assistance or where tasks need to be redistributed to maintain balance and productivity within the team.

1.2.2 Our Key Concepts in Detail

As part of our rigorous quality assurance process, we've developed a comprehensive checklist to guide our code review procedures. This checklist serves as a structured framework for evaluating each team member's contributions, ensuring that our code meets the highest standards of quality, readability, and functionality. Let's delve into each point in more detail:

Documentation: We assess whether the team member has adequately documented their code. This includes not only Javadoc comments for methods but also inline comments where necessary to clarify complex logic or algorithms.
Code Structure: We verify if the team member has adhered to the agreed-upon code structure. Consistency in code structure enhances readability and maintainability, making it easier for team members to navigate and understand the codebase.
Conventions: We check if the team member has followed the coding conventions established by our team. Consistent adherence to coding conventions promotes uniformity across the codebase and facilitates collaboration among team members.
Usable Code: We ensure that the code written by the team member is practical and usable. This involves verifying that the code fulfills its intended purpose and contributes meaningfully to the overall functionality of the project.
Logical Structure: We evaluate whether the team member has organized the code in a logical and intuitive manner. A well-structured codebase enhances maintainability and scalability, allowing for easier troubleshooting and future enhancements.
Exception Handling: We consider whether the team member has anticipated potential exceptions and implemented appropriate error-handling mechanisms. This involves both throwing exceptions when necessary and handling them gracefully to prevent unexpected crashes or errors.
Exception Handling (continued): We verify that the team member has implemented robust exception handling by catching exceptions where appropriate. This ensures that the application maintains stability and resilience even in the face of unexpected errors.
Unit Tests: We check if the team member has written unit tests for their code. Unit tests are essential for validating the correctness of code and detecting regressions. We ensure that unit tests cover the most critical methods in the class and also address exception scenarios effectively.

Following the review, we provide timely feedback to the developer. If any issues are identified, we collaborate to address them promptly, whether it involves fixing the code immediately or providing guidance for improvement in future iterations. This iterative feedback loop fosters continuous improvement and ensures the ongoing refinement of our codebase.

1.2.3 Bug Control

Within our project management toolkit, we maintain a meticulously curated ticketing list specifically dedicated to tracking and addressing bugs within our software. This comprehensive system, housed within a Google Sheet accessible to our team members, serves as our go-to resource for managing and prioritizing bug fixes.

1) Centralized Bug Tracking: Our Google Sheet serves as a centralized repository for logging all identified bugs and issues within the software. Each bug is meticulously documented, providing essential details such as the nature of the issue, its severity, steps to reproduce, and any relevant screenshots or error logs.
2) Priority-Based To-Do List: The ticketing list doubles as our prioritized to-do list for addressing bugs. Bugs are categorized based on their severity and impact on the software's functionality. High-priority bugs that pose significant risks or impair critical features are promptly escalated to the top of the list, ensuring they receive immediate attention and resolution.

3) Collaborative Issue Resolution: Team members collaborate seamlessly within the ticketing list, updating bug statuses, assigning ownership, and documenting progress in real-time. This transparency fosters effective communication and accountability, empowering team members to work together towards timely bug resolution.
4) Detailed Bug Descriptions: Each bug entry is accompanied by a detailed description, providing context and clarity to facilitate efficient troubleshooting and resolution. Clear and concise bug descriptions streamline the debugging process, enabling team members to quickly grasp the nature of the issue and devise appropriate solutions.
5) Long-Term Bug Management: The ticketing list also serves as a strategic tool for long-term bug management. More time-consuming or complex bugs are meticulously documented and prioritized within the list, ensuring they remain on our radar until resolution. This proactive approach prevents critical issues from slipping through the cracks and helps us allocate resources effectively to tackle persistent challenges.
6) Continuous Improvement: As bugs are addressed and resolved, the ticketing list serves as a historical record of our software's evolution. We leverage insights gained from past bug fixes to identify patterns, refine our development processes, and implement preventive measures to mitigate future occurrences.

Our ticketing list for bug tracking and resolution is a fundamental component of our software development workflow. By providing a centralized platform for documenting, prioritizing, and resolving bugs, it enables us to maintain the stability, reliability, and quality of our software product over time.

2 Analytical Quality Management

2.1 Analytical Procedure

2.1.1 QA Responsibles

Riccardo and Anisja are our QA-police, ensuring the quality of our code. They carefully review every important snippet, focusing on logic, structure, and coding standards. Their attention to detail guarantees our code's reliability. They then list any bugs in the Google Sheet and handle them themselves or report it to the developer, providing feedback for improvement.

2.1.2 All are responsible

As part of our continuous improvement efforts, we've implemented a structured practice of reviewing the most critical sections of our codebase during every second team meeting. This proactive approach allows us to foster a deeper understanding of each other's contributions and ensures that all team members are familiar with the key aspects of our software development efforts. During these code review sessions, we focus on examining the most important parts of our code, such as core functionalities, complex algorithms, and critical modules. By prioritizing these areas, we ensure that our collective knowledge and understanding are aligned with the foundational aspects of our project.

This practice serves several important purposes:

Knowledge Sharing: By regularly reviewing key sections of the code, team members gain insights into each other's work and coding practices. This knowledge sharing enhances colla-

boration and facilitates cross-functional understanding, enabling team members to support each other more effectively.

Quality Assurance : Regular code reviews help us maintain the quality and consistency of our codebase. By collectively examining critical sections of code, we can identify potential issues, refactor code as needed, and ensure that best practices are followed consistently across the project.

Risk Mitigation: By staying informed about the most important parts of our codebase, team members are better equipped to identify and address potential risks or dependencies. This proactive approach helps mitigate the risk of technical debt, software bugs, and performance issues.

Continuous Improvement: Code review sessions provide valuable opportunities for feedback and improvement. By discussing and analyzing our code together, we can identify areas for optimization, share insights, and collectively brainstorm solutions to challenges or complexities.

Overall, our practice of reviewing critical sections of code during team meetings reflects our commitment to collaboration, quality, and continuous improvement. By staying informed and engaged with the core aspects of our software development efforts, we ensure that our project remains on track, resilient, and poised for success.

3 Appendix

Ideas for even more QA: Using the `mockito` library, since it simplifies the server-client process: we do not need to have server and several clients to test this.