University:

UNIVERSIDADE DA CORUÑA

Department:

CAMPUS INDUSTRIAL

Erasmus Program:

SUSTAINABLE SHIP AND SHIPPING 4.0 (SEAS 4.0)

Course Title:

INDUSTRY 4.0 Enabling Technologies

Project/Report Title:

LAB 6 – SMART CONTRACT

Group Number:

ROBOT CARE

No.	Student Name	EMAIL ADRRESS
1	CABELO OQUEÑA, JORGE FRANCISCO	j.cabellooquena@studenti.unina.it
2	ALGAZZAR, ALMOATAZ MOHAMED MOHAMED KOTB	almoataz.algazzar@udc.es
3	ALLABERDIYEV, ATAGELDI	a.allaberdiyev@udc.es

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

The world is rapidly moving towards automation and smart contracts have emerged as one of the most promising technologies for automating business processes. A smart contract is a self-executing contract that is written in code and stored on a blockchain. It can automatically trigger actions, verify conditions, and enforce agreements between parties. In this project, we propose a smart contract that automates three tasks related to logistics, security, and maintenance.

The first task entails managing the supply chain, where the replenishment of raw materials is required upon depletion. The system will send requests for additional raw materials, which will be stored and transported using logistical procedures to ensure uninterrupted production.

The second task pertains to the monitoring of personnel movement within the robot area. The system will deploy a counting mechanism to track the number of individuals in the vicinity of the robot. This will enable effective management of the area around the robot and reduce the risk of accidents.

Lastly, the project incorporates a reminder system for the operators to lubricate the robot's joints every period. This serves as a preventive measure to avoid breakdowns and maintain the robot's operational capacity.

In this report, we will discuss the motivation behind this project, define the use case and objectives, describe the design aspects of the proposed smart contract, discuss the implementation details, present the tests and validation results, draw conclusions, and discuss the lessons learned. By the end of this report, the reader will have a clear understanding of the main idea of the project implemented in smart contract

2. MOTIVATION

The motivation for this project is to create an automated and efficient system that can handle the logistics, security, and maintenance aspects of a facility. By using a smart contract, the system can automate these processes, reducing the need for manual intervention and potential errors. The proposed smart contract is designed to send messages to the logistics department for timely material procurement, count the number of people detected by the ultrasonic sensor to maintain security, and remind the operator of the need for lubrication of the joints. The project aims to enhance the overall functioning of the facility, reduce costs, and increase productivity by streamlining the workflow. Furthermore, the smart contract can be customized and adapted to various settings, making it a versatile solution for different industries.

3. USE CASE DEFINITION AND OBJECTIVE

The objective of this project is to develop a smart contract that can automate three different tasks for a robotic production line.

Robot Care is an Ethereum smart contract designed to monitor and automate tasks related to the care and maintenance of a robot. The smart contract has three main functions:

1. Material Request: The smart contract sends a message to logistics every period requesting more materials.
2. People Counting: The smart contract counts people detected by the ultrasonic sensor that are not allowed to go inside the robot area.
3. Lubrication Reminder: The smart contract displays a message to the operator every period reminding them to lubricate the joints.

These three functions aim to increase the efficiency and reliability of robot care, reduce the risk of errors and accidents, and ensure optimal performance of the robot.

The Material Request function sends a message to logistics every 10 units of the final product produced. This function ensures that the robot always has enough materials to keep working without interruption, avoiding production delays or downtime.

The People Counting function counts the number of people detected by the ultrasonic sensor that are not allowed to go inside the robot area. This function ensures the safety of people by preventing unauthorized access to the robot area, reducing the risk of accidents or injuries.

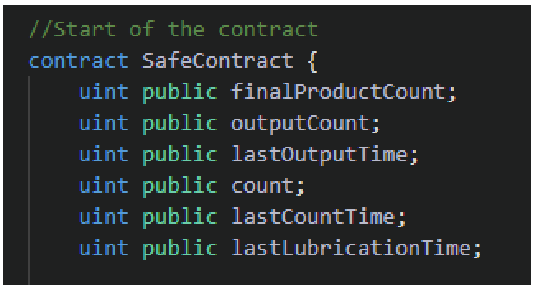
The Lubrication Reminder function displays a message to the operator every 20 units of final product produced, reminding them to lubricate the joints. This function ensures the proper maintenance of the robot, reducing the risk of wear and tear, and prolonging the life of the robot.

The Robot Care smart contract can be used in different settings where a robot needs to be monitored and maintained regularly, such as manufacturing plants, warehouses, or hospitals. The smart contract provides a reliable and automated solution for robot care, reducing the workload and potential errors of human operators, and ensuring the optimal performance and safety of the robot.

4. DESIGN ASPECTS OF THE PROPOSED SMART CONTRACT

The design aspects of the proposed smart contract are the following:

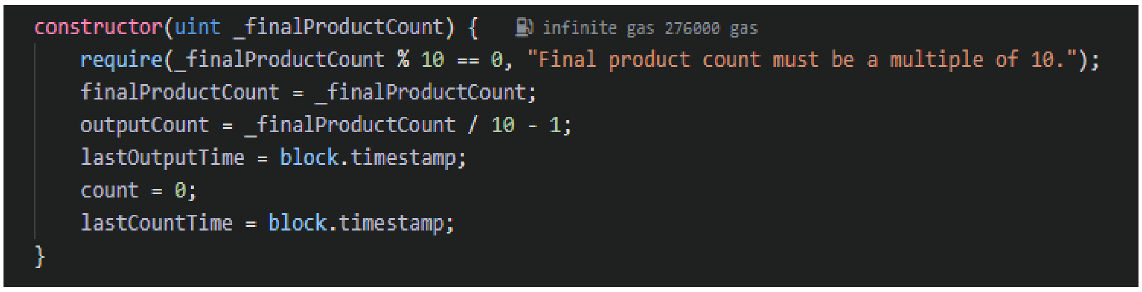
1. Contract Variables: The smart contract contains several variables that are used to store the state of the system. These variables include the final product count, the output count, the last output time, the count, the last count time, and the last lubrication time.



```
//Start of the contract
contract SafeContract {
    uint public finalProductCount;
    uint public outputCount;
    uint public lastOutputTime;
    uint public count;
    uint public lastCountTime;
    uint public lastLubricationTime;
```

Figure 1: Contract Variables

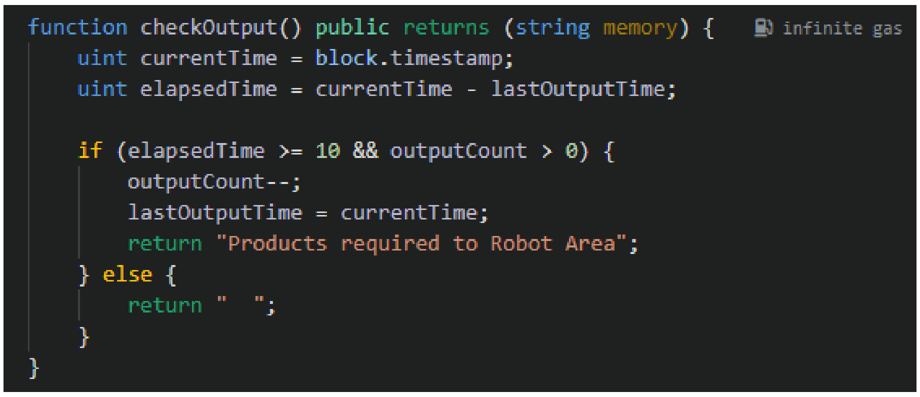
2. Constructor Function: The constructor function initializes the smart contract by setting the final product count and calculating the output count based on the final product count.



```
constructor(uint _finalProductCount) {    infinite gas 276000 gas
    require(_finalProductCount % 10 == 0, "Final product count must be a multiple of 10.");
    finalProductCount = _finalProductCount;
    outputCount = _finalProductCount / 10 - 1;
    lastOutputTime = block.timestamp;
    count = 0;
    lastCountTime = block.timestamp;
}
```

Figure 2: Constructor Function

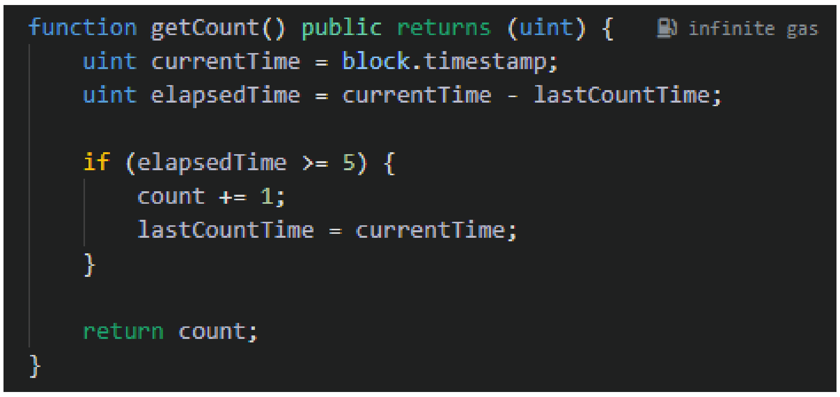
3. checkOutput Function: This function checks whether it is time to output products to the robot area. If it is time, the function decrements the output count and updates the last output time. Otherwise, the function returns an empty string.



```
function checkOutput() public returns (string memory) {    infinite gas
    uint currentTime = block.timestamp;
    uint elapsedTime = currentTime - lastOutputTime;

    if (elapsedTime >= 10 && outputCount > 0) {
        outputCount--;
        lastOutputTime = currentTime;
        return "Products required to Robot Area";
    } else {
        return "  ";
    }
}
```

Figure 3: checkOutput Function

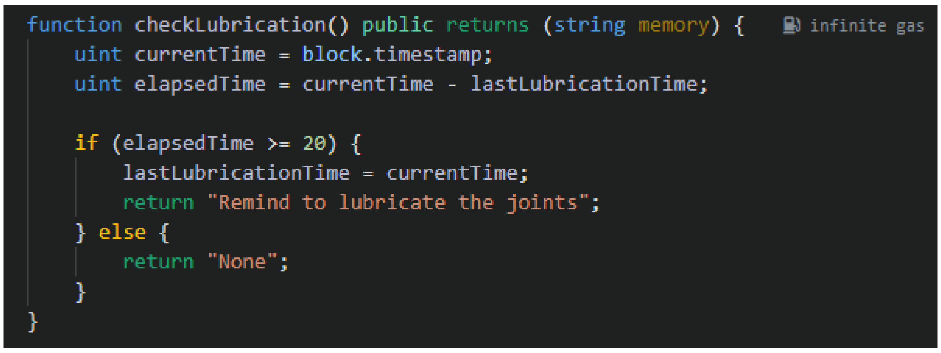
4. getCount Function: This function counts the number of people detected by the ultrasonic sensor every five seconds and returns the count.



```
function getCount() public returns (uint) {   infinite gas
    uint currentTime = block.timestamp;
    uint elapsedTime = currentTime - lastCountTime;

    if (elapsedTime >= 5) {
        count += 1;
        lastCountTime = currentTime;
    }

    return count;
}
```

Figure 4: getCount Function

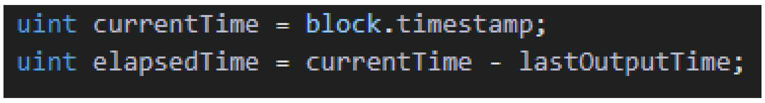
5. checkLubrication Function: This function checks whether it is time to remind the operator to lubricate the joints. If it is time, the function updates the last lubrication time and returns a message reminding the operator to lubricate the joints. Otherwise, the function returns "None".



```
function checkLubrication() public returns (string memory) {   infinite gas
    uint currentTime = block.timestamp;
    uint elapsedTime = currentTime - lastLubricationTime;

    if (elapsedTime >= 20) {
        lastLubricationTime = currentTime;
        return "Remind to lubricate the joints";
    } else {
        return "None";
    }
}
```

Figure 5: checkLubrication Function

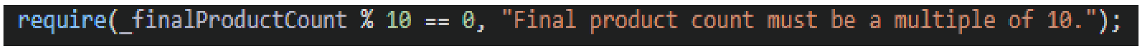
6. Timestamps: The smart contract uses timestamps to keep track of the time when events occur. This allows the smart contract to enforce timing constrains.



```
uint currentTime = block.timestamp;
uint elapsedTime = currentTime - lastOutputTime;
```

Figure 6: Timestamps

7. Error Handling: The smart contract includes error handling to ensure that the final product count is a multiple of 10.



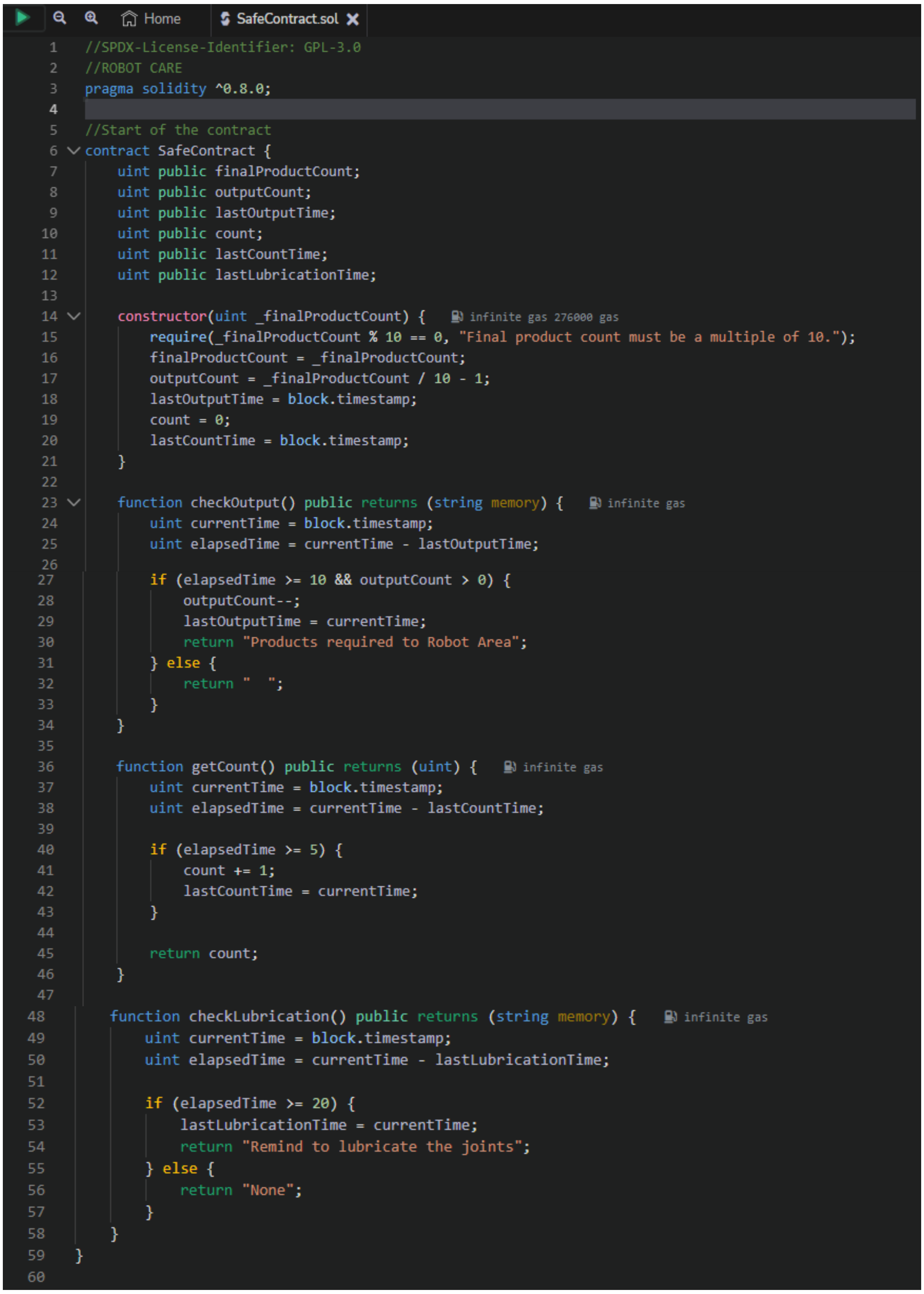
```
require(_finalProductCount % 10 == 0, "Final product count must be a multiple of 10.");
```

Figure 7: Error Handling

Overall, the design aspects of the proposed smart contract ensure that the system operates as intended, with proper timing of events and error-free execution.

5. IMPLEMENTATION

The implementation of the smart contract was done in Remix – Ethereum IDE as follows in the figure:

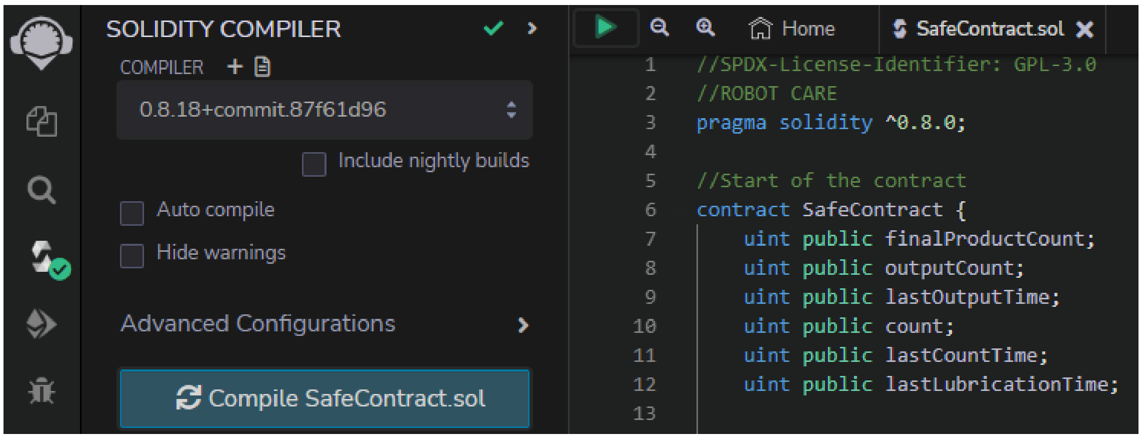


```solidity
//SPDX-License-Identifier: GPL-3.0
//ROBOT CARE
pragma solidity ^0.8.0;

//Start of the contract
contract SafeContract {
    uint public finalProductCount;
    uint public outputCount;
    uint public lastOutputTime;
    uint public count;
    uint public lastCountTime;
    uint public lastLubricationTime;

    constructor(uint _finalProductCount) {
        require(_finalProductCount % 10 == 0, "Final product count must be a multiple of 10.");
        finalProductCount = _finalProductCount;
        outputCount = _finalProductCount / 10 - 1;
        lastOutputTime = block.timestamp;
        count = 0;
        lastCountTime = block.timestamp;
    }

    function checkOutput() public returns (string memory) {
        uint currentTime = block.timestamp;
        uint elapsedTime = currentTime - lastOutputTime;

        if (elapsedTime >= 10 && outputCount > 0) {
            outputCount--;
            lastOutputTime = currentTime;
            return "Products required to Robot Area";
        } else {
            return "  ";
        }
    }

    function getCount() public returns (uint) {
        uint currentTime = block.timestamp;
        uint elapsedTime = currentTime - lastCountTime;

        if (elapsedTime >= 5) {
            count += 1;
            lastCountTime = currentTime;
        }

        return count;
    }

    function checkLubrication() public returns (string memory) {
        uint currentTime = block.timestamp;
        uint elapsedTime = currentTime - lastLubricationTime;

        if (elapsedTime >= 20) {
            lastLubricationTime = currentTime;
            return "Remind to lubricate the joints";
        } else {
            return "None";
        }
    }
}

```

Figure 8: Implementation in Remix - Ethereum IDE

6. TESTS AND VALIDATION

The validation of the code in Remix was done by compiling it.


SOLIDITY COMPILER

COMPILER: 0.8.18+commit.87f61d96

Include nightly builds

Auto compile

Hide warnings

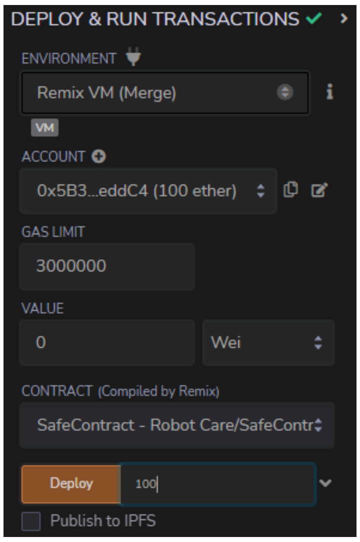
Advanced Configurations

Compile SafeContract.sol

```
//SPDX-License-Identifier: GPL-3.0
//ROBOT CARE
pragma solidity ^0.8.0;

//Start of the contract
contract SafeContract {
    uint public finalProductCount;
    uint public outputCount;
    uint public lastOutputTime;
    uint public count;
    uint public lastCountTime;
    uint public lastLubricationTime;

```

Figure 9: Validation of the code in Remix - Ethereum IDE

The following figures show test computed on the code in Remix for the 3 tasks.


DEPLOY & RUN TRANSACTIONS

ENVIRONMENT: Remix VM (Merge)

ACCOUNT: 0x5B3...eddC4 (100 ether)

GAS LIMIT: 3000000

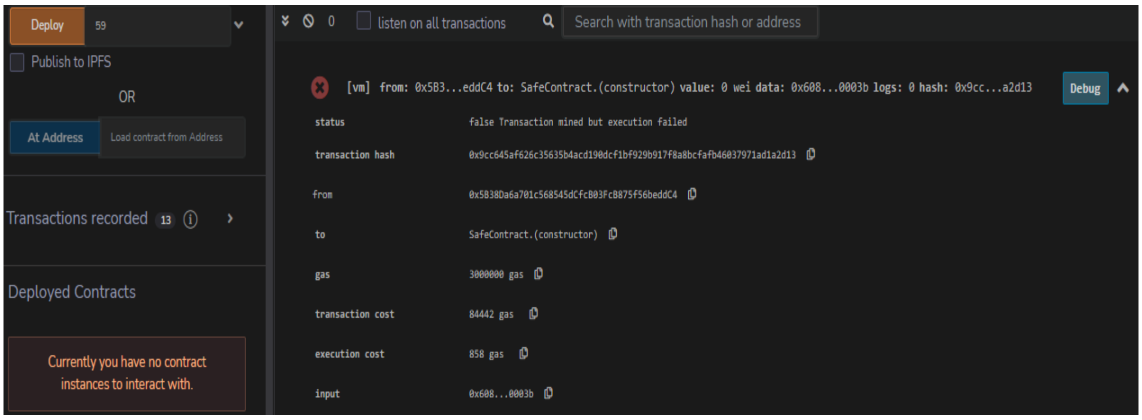
VALUE: 0 Wei

CONTRACT (Compiled by Remix): SafeContract - Robot Care/SafeContr

Deploy 100

Publish to IPFS

Figure 10: Input of the Final Product Count


Deploy 59

Publish to IPFS

OR

At Address | Load contract from Address

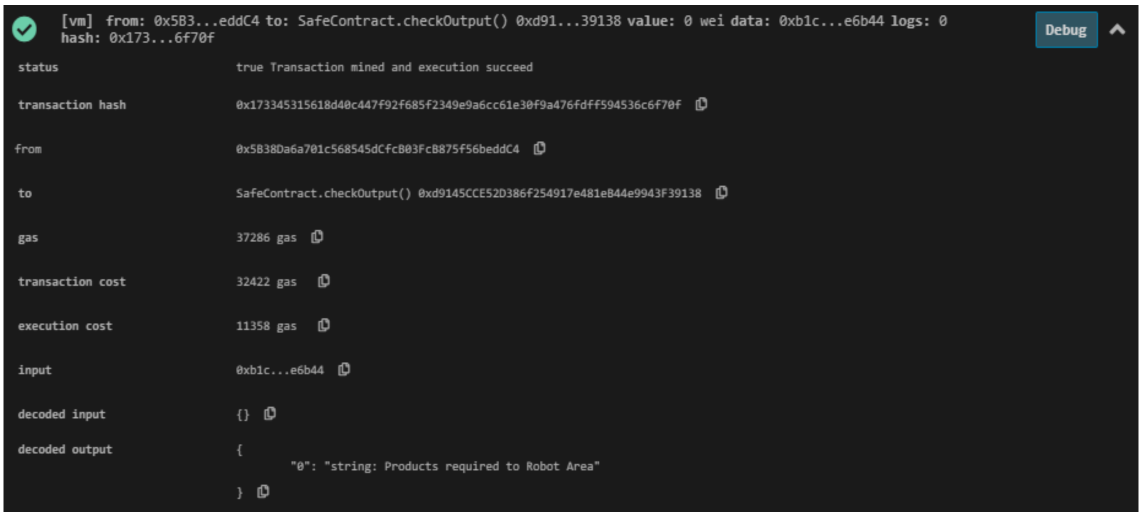
Transactions recorded 13

Deployed Contracts

Currently you have no contract instances to interact with.

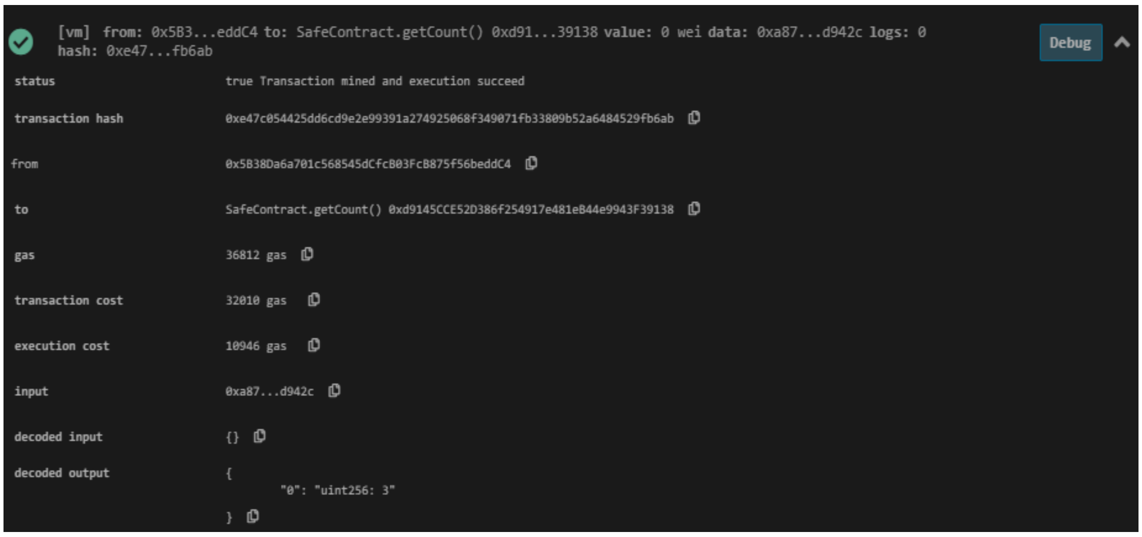
```
[vm] from: 0x5B3...eddC4 to: SafeContract.(constructor) value: 0 wei data: 0x608...0003b logs: 0 hash: 0x9cc...a2d13
status              false Transaction mined but execution failed
transaction hash    0x9cc645af626c35635b4acd190dcf1bf929b917f8a8bcfafb46037971ad1a2d13
from                0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                  SafeContract.(constructor)
gas                 3000000 gas
transaction cost    84442 gas
execution cost      858 gas
input               0x608...0003b
```

Figure 11: Error message in case of input not multiple to 10



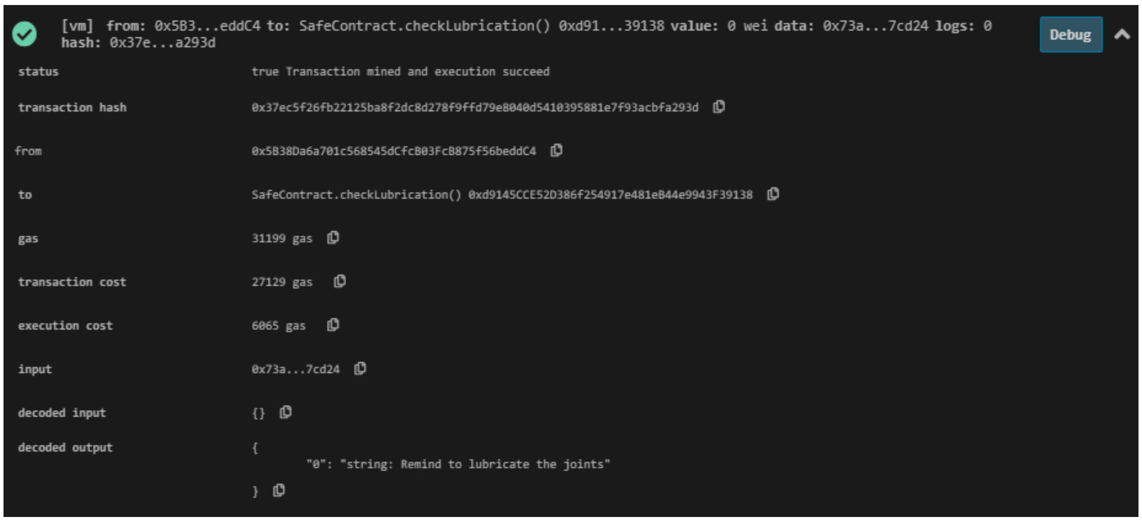
```
[vm] from: 0x5B3...eddC4 to: SafeContract.checkOutput() 0xd91...39138 value: 0 wei data: 0xb1c...e6b44 logs: 0
hash: 0x173...6f70f
status                  true Transaction mined and execution succeed
transaction hash        0x173345315618d40c447f92f685f2349e9a6cc61e30f9a476fdff594536c6f70f
from                    0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                      SafeContract.checkOutput() 0xd9145CCE52D386f254917e481eB44e9943F39138
gas                     37286 gas
transaction cost        32422 gas
execution cost          11358 gas
input                   0xb1c...e6b44
decoded input           {}
decoded output          {
                                "0": "string: Products required to Robot Area"
                        }
```

Figure 12: Status of checkOutput Function, requesting for products



```
[vm] from: 0x5B3...eddC4 to: SafeContract.getCount() 0xd91...39138 value: 0 wei data: 0xa87...d942c logs: 0
hash: 0xe47...fb6ab
status                  true Transaction mined and execution succeed
transaction hash        0xe47c054425dd6cd9e2e99391a274925068f349071fb33809b52a6484529fb6ab
from                    0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                      SafeContract.getCount() 0xd9145CCE52D386f254917e481eB44e9943F39138
gas                     36812 gas
transaction cost        32010 gas
execution cost          10946 gas
input                   0xa87...d942c
decoded input           {}
decoded output          {
                                "0": "uint256: 3"
                        }
```

Figure 13: Status of getCount Function



```
[vm] from: 0x5B3...eddC4 to: SafeContract.checkLubrication() 0xd91...39138 value: 0 wei data: 0x73a...7cd24 logs: 0
hash: 0x37e...a293d
status                  true Transaction mined and execution succeed
transaction hash        0x37ec5f26fb22125ba8f2dc8d278f9ffd79e8040d5410395881e7f93acbfa293d
from                    0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                      SafeContract.checkLubrication() 0xd9145CCE52D386f254917e481eB44e9943F39138
gas                     31199 gas
transaction cost        27129 gas
execution cost          6065 gas
input                   0x73a...7cd24
decoded input           {}
decoded output          {
                                "0": "string: Remind to lubricate the joints"
                        }
```

Figure 14: Status of checkLubrication Function, reminding for lubrication

7. CONCLUSIONS AND LESSONS LEARNED

- Testing and validation of the smart contract have demonstrated its functionality and effectiveness, meeting the requirements of the proposed use case.
- The smart contract effectively automates the process of requesting more materials, counting people, and reminding operators to lubricate joints, reducing the need for manual intervention and improving efficiency.
- The use of Solidity language and Ethereum blockchain technology provides transparency, security, and immutability, ensuring that the contract's execution is tamper-proof.

8. REFERENCES

1. Prof. Paula Fraga Lamas, Lecture Notes: Industry 4.0 Enabling Technologies, UDC, Seas 4.0 2023.
2. Dr. Tiago M. Fernández Caramés, Lecture Notes: Industry 4.0 Enabling Technologies, UDC, Seas 4.0 2023.