POLITECNICO DI TORINO

Master's Degree in Engineering and Management

Master's Degree Thesis

Development of a Mobile App to improve the drivers' behavior exploiting a Gamification mechanism built on Blockchain

Supervisors

Prof. Valentina GATTESCHI

Prof. Alberto BUTERA

Candidate

Noemi ROMANI

April 2024

Acknowledgements

I would like to dedicate this space to the people who, with their support, helped me to reach this goal.

First of all, I would like to particularly thank my supervisor, Professor Valentina Gatteschi, for the trust she placed in me from day one and for her indispensable advice. I would like to also thank Alberto infinitely for his support, patience and availability.

A special thank you to Professor Marco Barla, for giving me the opportunity to calmly pursue my sporting career in parallel with my university career.

I infinitely thank my parents and my twin sisters, Michela and Chiara, who have always supported me, from near and far, supporting my every decision and allowing me to face my entire studies peacefully. I would like to thank Stefano for always being a great supporter of mine, in sport and in life, and for allowing me to chase my dreams. Thanks to Giorgia, for being by my side throughout much of the journey. And thanks to my brother-in-law Davide, for always being beside me in silence, never failing to give me his message and support on every important occasion, in sports tournaments and university exams.

A special thank you to my grandmother, I always felt you by my side.

I dedicate a final thank you to myself, to my ambition and perseverance which have never made me lose sight of my goals. To my perseverance, which has allowed me to aim high in sport and study. I thank the bad moments, all the defeats in sport and in life, because it is in those moments that I brought out the strength that I didn't think I had, that I tried with all of myself to look for the ray of sunshine in the storms, and that I have learned to believe that in life, as in sport, you either win or you learn.

Noemi

Summary

It is well known that auto Insurance Companies (ICs) use driving-related data collected by continuously tracking the drivers' behaviors to determine and adjust the auto premiums defining the Personalized Car Insurances. It is also well known that the majority of traffic incidents is due to human factors, especially to aggressive driving. However, the rules of Personalized Car Insurances, the evaluation model according to which auto premiums are calculated as well as the process through which the drivers' behavior is detected by the ICs, are unknown. In this work, recognizing the importance of monitoring the drivers' behavior and considering the necessity to overcome the limits of the actual Personalized Car insurances, a system based on a developed Blockchain solution and a developed Mobile App solution is proposed. Specifically the Mobile app is used to implement the process of detecting the drivers' behavior: at first, raw data acquisition and transformation is executed; then, a developed algorithm is implemented for analysing data with the aim to detect aggressive events; at the end, a score is computed by implementing an Event-count based algorithm. The data considered to be relevant for classifying the drivers' behavior are then transferred and stored on the Blockchain platform. The blockchain solution describes a Gamification mechanism, which logic is defined by a developed Smart Contract: according to a final ranking, the drivers resulting to have the safest driving style are awarded. The mutual contribution of these two system's components allows to monitor the drivers' behavior by relying on transparent processes and data, by preserving the privacy of sensitive and personal data, and by stimulating the drivers toward a safe driving style through a Gamification context.

Table of Contents

List of Tables VI

List of Figures VII

1 Introduction 1

1.1 Thesis Structure 4

2 State of Art 5

2.1 The Blockchain technology 5

2.1.1 What are blocks, and how are they created? 6

2.1.2 The blockchain network and the different blockchain models 7

2.1.3 Cryptography 8

2.1.4 Consensus protocols 9

2.2 PEST Analysis 11

2.2.1 Political 11

2.2.2 Economic 12

2.2.3 Social 12

2.2.4 Technological 13

2.3 Some Blockchain applications 13

2.3.1 Financial Services 13

2.3.2 Industry 4.0 14

2.4 Analysis of the Thesis's Use Case 14

2.4.1 Technology's Key Elements 15

2.4.2 Technological Paradigms 16

2.4.3 Analysis of Paradigms through KPIs 17

2.4.4 Radical and disruptive innovation 20

2.4.5 Technology's Development process 20

2.4.6 Diffusion of the innovation in the market 23

2.5 Gamification in the Insurance sector 25

3 System Architecture 28
3.1 Technologies . . . 28
3.2 Architecture . . . 29
3.3 Process of detecting the drivers' behavior . . . 32
3.3.1 Sensors' raw data acquisition and data transformation . . . 32
3.3.2 Data transformation . . . 35
3.3.3 Classification of driving events . . . 37
3.3.4 Score's Computation Process . . . 44
3.4 Service for Location Update . . . 47
3.4.1 Location Service as Binding Service . . . 47
3.4.2 Location Service as Foreground Service . . . 49
3.4.3 Location Update Request . . . 51
3.4.4 Geocoding Process . . . 53
3.4.5 Map's Integration . . . 53
3.4.6 Device's Orientation . . . 54
3.5 Blockchain . . . 56
3.5.1 Interaction with Blockchain . . . 56
3.5.2 Smart Contract . . . 58
3.5.3 System Model . . . 58
3.5.4 Gamification Context . . . 61

4 User Experience 64
4.1 IDEF0 diagrams . . . 64
4.2 UI Level . . . 66
4.2.1 UML: Activity Diagrams . . . 68
4.2.2 Landing Page . . . 69
4.2.3 TripsActivity Page . . . 69
4.3 Data Layer . . . 78
4.3.1 UML: ER Diagram . . . 78
4.4 UML: Sequence Diagrams . . . 81

5 Results 88
5.1 Testing the Smart Contract . . . 90

6 Conclusions and future works 93

Bibliography 95

List of Tables

2.1 Comparison between PoW & PoS 10

3.1 Comparison about variables' visibility between the Mobile App and the Smart Contract 61
3.2 Smart Contract's Functions 63

5.1 Cost analysis of the main functions on Ethereum and Polygon . . . 92

List of Figures

2.1 Present Technological Paradigm: Traditional PCI 16
2.2 Emergent Technological Paradigm: PCI integrated with Smart Contract 16
2.3 Performance and Diffusion s-curves. 22
2.4 Ethereum: Daily Transactions' Volumes 22
2.5 Cloud market size worldwide in 2021, with a forecast for 2030 . . . 23
2.6 Market segments along the technological life-cycle 24

3.1 System Architecture 30
3.2 Java Classes 31
3.3 Process of detecting the drivers' behavior 33
3.4 Sensor's coordinate system. About the device's orientation, it is important to remark that, even if the device's orientation is changing, the disposition of the axes remains the same. Then, the same readings can be detected on the same axes, independently on whether the smartphone is kept horizontally or vertically. 34
3.5 Orientation of sensor's coordinate system in vehicles 38
3.6 Table with reference thresholds 39
3.7 Harsh u-turn event 45

4.1 IDEF0: GoSafety 65
4.2 IDEF A-0: GoSafety functionalities 66
4.3 Activity Diagram: Landing Page 70
4.4 Landing Page Layout 71
4.5 Login Page layout 72
4.6 Sign Up layout 72
4.7 Activity Diagram: TripsActivity 75
4.8 TripsActivity layout, with Fragments 76
4.9 Drawer menu layout 76
4.10 Activity Diagram: Drawer Menu 78
4.11 ER Diagram 79

4.12 ER Diagram: Smart Contract 80
4.13 UML Diagram Solidity 83
4.14 Sequence Diagram 84
4.15 Sequence Diagram 85
4.16 Sequence Diagram 86
4.17 Smart Contract Sequence Diagram 87

5.1 Etherscan: successful transaction executed 91

Chapter 1

Introduction

The objective of this Thesis was to incentivize drivers toward a safe driving style while guaranteeing transparency and privacy-preservation in the analysis of driving-related data and evaluation process. This Thesis work consisted of developing a *Mobile App* to be used to collect and process driving-related data with the aim of detecting eventual harsh-events occurring during drivers' trips and of determining a score for each trip, and to develop a *Smart Contract* defining a Gamification mechanism built on a Blockchain platform, aimed at awarding the safest drivers at the end of a game and at storing the relevant data.

It is well known that auto Insurance Companies (ICs) use driving-related data collected by continuously tracking the drivers' behaviors to determine and adjust the auto premiums defining the Personalized Car Insurances. The adjustment of auto premiums is based on the evaluation results. Anyway, one of the barriers of this Insurance model is the lack of transparency: the rules of Personalized Car Insurances are not clear and the evaluation model according to which auto premiums are calculated and revisited, based on the drivers' behavior, is unknown. In addition, even if the objective of detecting the drivers' behavior is known and goes beyond the Insurance field, the process through which the drivers' behavior is detected by the ICs is unknown. According to existing research reports [1], the majority of traffic incidents is due to human factors, especially to aggressive driving. A reduction of aggressive events, induced by monitoring the driving style, could lead to a lower number of road traffic incidents, and could also result in a lowering of vehicle consumption and gas emissions, as aggressive driving has been estimated to increase fuel consumption by around 40% [1]. Believing in the importance of monitoring the drivers' behavior and considering the necessity to overcome the limits of the actual Personalized Car insurances in terms of data and processes transparency, this work focused on the development of a Blockchain solution from one side and of a Mobile App solution from the other side.

A considerable number of existing algorithms can be implemented in the process of detecting a driver's behavior [2], which is usually structured in three phases. First, sensors raw data are detected and transformed which implies cleaning signals in order to extract noises. Then, the signal is used for classifying driving events, knowing that the classification can be either binary (aggressive or non-aggressive event) or multi-class (identifying the specific driving feature leading to the aggressive event). At the end of the process, the detected and classified events are integrated in the computation of a score. There are many aspects making difficult to individuate the 'best' algorithm, due to the shortage of datasets created in real driving conditions, and due to the fact that the detected data can be affected by many factors, i.e. the driving environment, the device used for data recording. Starting from these considerations [2], the developed Mobile App proposes a process of detecting the drivers' behavior based on the adoption of the smartphone as device to collect driving-related data and on algorithms implemented for the transformation of data through the application of filters, for the analysis of data based on reference parameters derived from existing research works [1] focused on the evaluation of the different available algorithms, for the computation of scores according to the events detected.

The adoption of the smartphone for collecting driving-related data derived from the growing market for smartphone applications as recorders and processors of driving-related data: the advantages of smartphones and similar devices is that, through the sensors they are already equipped with, could be used like black-boxes (the traditional way to record drivers' data) without additional costs for the hardware equipment [1]. One of the other advantages related to the smartphone's use is the possibility to process data on-board and the access they have to communication networks for data transfer [1].

To address the issue of data transparency, the proposed Mobile App also implements algorithms for the storage of sensitive and relevant data, that have been designed and programmatically developed so that users do not have the chance to select, to choose or to cheat data to be transferred to the Blockchain. Furthermore, the evaluation model is clearly explained to the users presented in a specific section of the App's drawer menu.

Moving to the Blockchain solution that has been designed for addressing the issue of transparency related to the Personalized Car Insurances, it is well known that a Blockchain platform is an online ledger. It implies that all the data written on the Blockchain platform are transparent and immutable. The project's proposed solution was conceived to store on the Blockchain the data relevant to the evaluation process of the driver's behavior, in a way that they can be visible to everyone and that cannot be modified or removed. Furthermore, the proposed Blockchain solution is based on a Smart Contract which transparently defines the logic of a Gamification Context. If in the real Insurance context, the drivers' behaviors can be stimulated to approach a safe style by applying lower insurance fees and through auto-premiums adjustments, in this thesis work a Gamification context has been designed. The application's users, by calling some methods of the Smart Contract, become Players. When signing up for the game, a fund is transferred from their wallet address to the Smart Contract. At the time of game closing, a final ranking of the players is determined: the players ranked in the first three places are awarded with a 'prize money', defined as percentage of the total amount the players sent to the Smart Contract for that game.

It is well known that among the main challenges of the Blockchain technology, there are the concepts of privacy-preservation and of data consistency. With the aim of addressing those issues, the two proposed solutions have been developed to give a mutual contribution. Data transferred to the Blockchain from the Mobile App only relate to the scores achieved by the application's users at the end of a car travel. As the driving features determining that result are not specified on the Blockchain platform, it could be possible to associate this lack of relevant and sensitive data to a sort of data inconsistency. Behind this decision, there was the concept of data privacy. At the same time, in order to fulfill the project's goals of data consistency and data transparency, all the sensitive data which are relevant to the process of score's determination are visible and accessible to the user via Mobile App.

As briefly described, the main project's design goals were:

- data and process transparency;
- data privacy-preservation;
- data consistency;
- incentive for drivers toward a safe driving style.

The main implementations carried out as the subject of the thesis were:

- design of the entire system model, and of the Gamification Context to achieve the design goals of the project;
- development of the Mobile App's backend, for detecting and processing driving-related data, and for the interaction with the Smart Contract;
- development of the Mobile App's frontend of the Mobile App, for a mobile user and for making interesting the user experience;
- development of the Smart Contract for defining the Gamification's logic and managing transfer of funds.

1.1 Thesis Structure

- Chapter 2 – State of Art: the first part is dedicated to the description of the Blockchain technology, its basic elements, the governance, advantages and disadvantages of existing systems. The second part starts with the PEST Analysis, provides some examples of the blockchain technology's application in sectors different from the Insurance one, and then goes in detail about the use-case analyzed in this work, comparing the present technological paradigm, intended as traditional Personalized Car Insurance, with the emergent one, intended as Personalized Car Insurance with the adoption of a Smart Contract. The last part is dedicated to the use of Gamification in the Insurance sector, with some examples given.
- Chapter 3 - System Architecture: this chapter provides a consistent description of the entire system model, going in detail on the implemented algorithms, thus defining the logic behind the project' components, and the technical aspect of the developed programming code.
- Chapter 4- User Experience: this chapter offers a different view of the work performed, by describing it by means of direct interaction of the user with the developed product. It moves from the User Interface to the Data Layer by describing the main components defining the interaction of a general user with the Mobile App through the support of IDEF0 diagrams and UML diagrams. A ER Diagram and a Sequence Diagram describing the Smart Contract logic are also given.
- Chapter 5 - Results: this chapters analyzes the achieved results.
- Chapter 6 - Conclusions: this final chapter draws conclusions regarding the developed Thesis and traces the lines for possible future developments.

Chapter 2

State of Art

The Blockchain technology is very often defined as "*The Internet of the future*", as it represents a real infrastructural upheaval with possible repercussions in a lot of different sectors. Its innovative component is linked to the possibility of developing decentralized applications, carrying out immutable transactions and removing the presence of intermediaries. Based on concepts of opens source and decentralization, this technology is already considered today one of the most innovative technologies available for companies. However, the spread and the adoption of Blockchain are slowed down by prejudices and a general feeling of mistrust, due primarily to disinformation and to all the risks of a new technology, still in phase of prototyping, and of a decentralized world, which, inevitably seems destined to be more and more real, possibly leading to a real world where is possible to keep track and control of data, in which authority goes from being concentrated on a few central entities to being equally distributed, in which the presence of guarantors and intermediaries is replaced by an incorruptible system on which results easier to have confidence.

This chapter opens with a brief overview of the technology, describing its main topics. Then, the core of the chapter is focused on the analysis of a specific use-case, the one for which this thesis work tries to forge and provide a Blockchain solution. Through this use-case a depth analysis of this new technology is carried out. At the end of this chapter, a brief description about the use of Gamification in the insurance sector, moving from the area of car insurance to the one of life insurance, is given.

2.1 The Blockchain technology

It's worth to start from an exhaustive definition [3], which includes most of the aspect characterizing such technology, and then moving to the main topics to give a brief overview of the blockchain system.

"*Blockchain is a digital ledger, decentralized and distributed on a network, structured as a chain of registers (blocks) responsible for storing data (from value transactions to entire digital applications). It is possible to add new blocks but is not possible to modify or remove blocks previously add to the chain. In this ecosystem, encryption and consensus protocols ensure security and immutability.*"

2.1.1 What are blocks, and how are they created?

"Blockchain is a digital ledger *structured as a chain of registers (blocks) responsible for storing data* (from value transactions to entire digital applications)" [3].

Blocks are data structures added to the blockchain sequentially, a block at a time. Each of them contains a mathematical proof generated through the cryptography's use, which ensures its sequentiality from the previous block, resulting in a chain of blocks. The connection between blocks is generated via the cryptographic hash function, which creates an indissoluble mathematical link between them.

The Hash function

This function is used to map data of arbitrary dimensions into data of fixed dimensions. So, if the input of a hash function can be almost anything (pdf file, mp3 file, work sheet, entire blockchain), the output, called *hash*, will always contain a fixed number of bits. The main aspects to be considered for a brief description of the hash functions are:

- The same input always produces the same output; the function's output is the hash, which takes the form of a string of letters and numbers; the hash is not saved in the block, but it is calculated whenever necessary.
- Even the slightest change in the input produces a dramatic change in the function's output.
- It is a one-way function: is computationally easy to generate the hash starting from any input, but there is no way to calculate the function's input starting from the hash; the unique way is the brute-force method which consists in trying all possible combinations.

Starting from these considerations, in case of attempts to add, modify or remove some of the data or information stored in any block, the hash of that block will be changed and also the hashes of all the subsequent blocks. This is due to the fact that, when a new block is added to the chain, the hash of the previous block is inserted into the input to generate the new block. In summary, every block contains some data, some information, and the hash of the previous block. As any change in one of the chain's blocks leads to a change in the hashes of that blocks and the subsequent ones, in order to evaluate the status of a Blockchain, to compare different versions of the same Blockchain, only the hash of the last block of the chain is sufficient.

2.1.2 The blockchain network and the different blockchain models

"Blockchain is a digital ledger, *decentralized and distributed* on a network."

Every machine connected to the Blockchain network is a node, which can be a full-node or a light-node. A full-node represents the hardest way to interact with a Blockchain but the most secure: it works independently, by propagating blocks and valid transactions, ignoring the invalid ones. Conversely a light-node represents the easiest way, but it requires a full-node to work: it does not verify data correctness, but simply works with the data received from a full-node. By considering the network's architecture, the network's logic and the network's authority, it's possible to describe the Blockchain's models: public, centralized, consortium. More specifically, the three models are characterized by the same network's architecture and logic:

- A Blockchain is *architecturally* decentralized, there is no single point of failure.
- A Blockchain is *logically* centralized; it is all the time characterized by a single logic status; there are different protocols for all the nodes to reach the same logic status, as defined in the section 2.1.4.

Conversely, in terms of authority, the three models can be distinguished:

- *Public* Blockchain: it is a permissionless system, where every node has equal rights and responsibilities, every transaction is public and analyzable; generally public Blockchains are also open-source, making public the code defining the Blockchain's regulation.
- *Private* Blockchain: this permissioned system sacrifices complete decentralization in exchange for control over access permissions and usually better performances. Not all the nodes can read/write data on the Blockchain, not all the nodes can participate in the transactions' verification process. This Blockchain model is generally adopted in the industrial sector.
- *Consortium* Blockchain: this is a permissioned system, where, unlike private Blockchain, control and authority are distributed over the network's nodes; it is a hybrid solution between the public model and the completely private model.

2.1.3 Cryptography

"In this ecosystem, *encryption* and consensus protocols ensure security and immutability."

Cryptography refers to the study of secure communication techniques in a hostile environment, such as Internet. Blockchain is a system where cryptography, more specifically public key cryptography, plays a particularly important role.

What are cryptographic keys?

The basic idea is to use a pair of keys, extremely large numbers usually represented in hexadecimal (0-9 to represent numbers from zero to nine, a-f to represent numbers from ten to fifteen), in mathematical relation to each other:

- a *private* key, randomly generated, which must remain secret;
- a *public* key, mathematically derived from the private key, which ca be shared to anyone.

It is computationally very easy to generate a public key starting from a private key; conversely, to derive the private key starting from the public one is almost impossible.

Why the public key cryptography?

Public key cryptography can be used to ensure encryption of information or data (encryption of messages through the use of the private key of the recipient, decryption of data through the use of the recipient's public key), authentication of users, integrity of data or information, and non-repudiation of data by users.

The main application of public key cryptography in Blockchain consists in the generation of wallet addresses and the authentication of transactions by mean of digital signatures.

- *Wallet addresses:* from a technical perspective, a wallet address is the output of a mathematical operation including public key cryptography and hashing.
  1. A private key is generated starting from a random number, through the hash function.
  2. The corresponding public key is derived from the private key, through a mathematical process.
  3. The public key passes through a series of cryptographic algorithms in order to obtain an address.

- *Digital signature:* digital signatures are the output of a combination of hashing and public key cryptography as well. Through a digital signature is possible to obtain:
  - *Authentication*: a private key is associated to a specific user; a digital signature unequivocally proves that data/a message/information have been sent by that user.
  - *Integrity*: once a message has been digitally signed, any attempt to modify the message makes the signature invalid.
  - *Non-repudiation:* when someone signs a message, he cannot, at a later time, deny having signed it.

  Starting from this brief introduction of wallet addresses and digital signatures, it is possible to understand their role in the creation of a transaction.

- *Transactions*: the sender digitally signs a transaction with the private key associated to the address used. The digital signature ensures that:
  - The address creating the transaction belongs to the user, i.e., when users want to send 1 bitcoin they need to have the private keys of an address with at least 1 bitcoin associated. The basic requirement for creating a transaction is having the object of the transaction. The transaction is signed with the private key of the user (*authentication*).
  - The transaction has not been modified after the signature (*integrity*).
  - The user, owner of the private key (used in the transaction) cannot deny to have signed the transaction (*non-repudiation*).

It's worth to specify that creating a transaction does not mean the transaction is valid. A transaction is valid only if approved by the network *consensus*, as explained in the next section.

2.1.4 Consensus protocols

In order for a transaction to be approved, the network nodes have to reach an agreement on a single state. The process, through which the network reaches a decision about what happened in the Blockchain, is called consensus. In a Blockchain system the consensus is an agreement on what happened, and it is the only possible truth about the current state of the Blockchain. However, the achievement of the consensus in a distributed and decentralized system remain a complex problem. Many algorithms were designed and tested to solve this problem: the most used algorithms in the Blockchain context are the *Proof of Work* (*PoW*) and the *Proof of Stake* (*PoS*). The nodes actively participating in the consensus process (by adding new blocks, by ensuring the transactions' validity) are called *miner* and the process is called *mining*. The mining process consists in validating, aggregating into blocks and adding transactions to the Blockchain. When a new block is added to the Blockchain, the miner is rewarded for the work done according to the rules of the Blockchain: the reward usually consists in the transaction fees of a block and, possibly, in the cryptocurrencies generated when a new block is added.

In order to describe and highlight the differences between the two protocols above mentioned, the following table is proposed [3]:

	PoW	**PoS**
Needs	Computational Power	Stake (cryptocurrencies and other parameters
Who is going to create a new block	Miner, chosen randomly according to the computational power	Validator, chosen in advance according to the own stake
Fairness of the system	Yes	Yes
Time required to generate a new block	Variable, depending on the time required to find a valid PoW solution	Fixed
Required Computational Power	Very high	Minimum
Reward to the miner	Transaction fees + possibly cryptocurrencies generated with the new block	Transaction fees + possibly cryptocurrencies generated with the new block

Table 2.1: Comparison between PoW & PoS

Focusing on the first line, it is possible to notify how the Proof of Work is a protocol used to reach a distributed consensus in which the voting power is based on the computational power. In the PoW-mining, the nodes of the network compete to solve a complex mathematical problem (reverse harsh with some constraints): the probability to solve this problem is very low and the only way to find a PoW valid solution consists in trying all the possible combinations until the right solution is found. In fact, in a system implementing the PoW, a block is valid only if contains a valid solution to the PoW; a PoW solution must satisfy a constraint called *difficulty* to be considered as valid. In the PoS-mining, even if the final objective is the same of the PoW algorithm, the process to reach the the solution in completely different. With respect to the Proof of Work, which rewards the miners who succeed in solving mathematical problems, the Proof of Stake algorithm uses and alternates *validators*, who are the equivalent of the miners. They are chosen in advance according to the own stake amount: basically, users owning a token amount can stake their own token in exchange for the right to confirm the block transactions (becoming *validators*) and to receive a reward. Every token corresponds to a vote.

Moving the focus to the second line of the table 2.1, it is interesting to notice that in the PoS the user who is going to be the validator is defined in advance, whereas in the PoW there is a real competition among nodes. With the aim to be the first, every miner starts to find a valid PoW solution: to each invalid solution a miner changes the value of a number, called *nonce*, which is used to vary step-by-step the input of the hash function until the resulting hash satisfies the difficulty value. By highlighting the importance of the computational power, which determines the miner's hashrate (number of hashes calculated in a second [H/s]), the probability for a miner to be the first in finding a PoW valid solution is:

$$Probability_{FirstMiner} = hashrate_{miner}/hashrate_{network} \tag{2.1}$$

As already written, in the PoS protocol the validator is chosen in advance according to the stake own. The probability for a user to be chosen as validator, then the user's voting power can be defined as:

$$Voting_{power} = Stake_{validator}/Stake_{network} \tag{2.2}$$

Regarding the third line of the table 2.1, it can be said that both the protocols ensure fairness toward users: in the PoW protocol, a miner owning 5% of the computational power an average wins the PoW and obtains the right to create a new block 5% of the time. In the PoS protocol, a validator owning 5% of the network stake an average obtains the right to create a new block (and to gain the reward) 5% of the time.

Finally, regarding the last line of the table 2.1, it is worth to specify that, before a block is added to the Blockchain and before a miner/validator gains a reward, the block is propagated to the network, waiting for some confirms of the previous blocks. The validating process is a backwards process where the previous blocks, by passing their hash in input to the hash function, can prove the validation of the new block's hash. Once a certain number of confirms is received, the block is added to the Blockchain and the miner/validator can be rewarded.

If comparing the two protocols, the PoS results to be more secure and less expensive. Furthermore, through the PoS is possible to infer economic disincentives to bad users, and to make users more stimulated in being loyal with the blockchain system.

2.2 PEST Analysis

In this section, an analysis of the Political, Social, Economic and Technological factors that could impact on the development process of the Blockchain technology and on the diffusion of Blockchain applications is carried out.

2.2.1 Political

The world of Blockchain is witnessing the birth of a growing number of projects in the most disparate sectors, but at the same time the world of cryptocurrencies is going through a phase of regulation in various Countries. While with regards to the adoption of the Blockchain technology there is no particular restriction to its use, with regards to the use of cryptocurrencies, each Country is defining its position with respect to the regulations regarding the use of cryptocurrencies as a means of payment, fundraising using token, exchange operation methods. By way of example, it is possible to identify a category of Countries *"predominantly in favor"* of cryptocurrencies, and in some cases with already defined regulations: it is possible to mention Switzerland, Malta, Estonia, and Japan. Conversely, it is possible to define a category of Countries *"predominantly against"* the use of cryptocurrencies: China expressly prohibits ICOs and fiat-crypto transactions on exchanges. As regards Italy, in September 2018 it joined the European Blockchain Partnership together with 26 other EU member States [4]. One of the objectives is to regulate cryptocurrencies and the investment methods based on them (ICO), together with a push for the implementation of such technology by enterprises.

2.2.2 Economic

Considering the role of Blockchain in the "web revolution" (*Web 3.0*), the numerous problems related to Web 2.0 (*read-write*) have led many people to believe that a structural revolution of the web is necessary, in order to bring it to its initial idea of a decentralized, open and universal platform. The merging need for re-decentralization of web services can be satisfied by implementing this new technology: it would grant decentralization and democratization of accesses, which is the opposite with respect to the monopoly of data enjoyed by the current web's giants; users would regain possession of data: in Web 2.0 data are given away for free in exchange for free services (furthermore, how data is used is often not transparent); the persistence of data would be guaranteed, avoiding the risk that, being saved on centralized databases, data would be censored, lost or deleted. A concrete examples of Web 3.0 services are Ethereum for Cloud Computing, the same service of Google Cloud or Amazon C2 provide in Web 2.0.

2.2.3 Social

Focusing on the social environment, limits related to the Web 2.0 led to rediscovered needs for decentralization, transparency of data/processes, persistence of data. These needs introduce the concept of *data sustainability*, perfectly integrated in this era, which is the era of sustainability par excellence, with the main concept extended to all the sectors. Focusing on the needs of data transparency, data possession, data persistence and decentralization, they can be met by the model of Blockchain Consortium, in which the consensus process is controlled by a-priori defined nodes over which authority is distributed. It grants transparency of data and processes, and, with respect to the public Blockchains, companies can decide to have control over data. In the industrial sector and more generally in processes that require collaboration between multiple institutions it is the most adopted

Blockchain model: it can be implemented to improve the efficiency of the supply chain process, in which a product requires the coordination of many different entities to move from producer to consumer, or it can be implemented by the European Union, which could use such technology as both a ledger and a system of voting, allowing each Country to represent a node.

2.2.4 Technological

Investments in the exploration and exploitation of the Blockchain technology are increasing everywhere, with particular focus on the cryptocurrencies and on the application of the Blockchain technology together with Smart contracts to the economic processes. By removing the presence of intermediaries and central authorities, those investments have led to a completely new concept to arise: *smart economy*. From the smart economy, the concepts of *smart property*, *digital identity*, *token model*, have been defined. A concrete example of Blockchain's application to economic systems are the Decentralized Autonomous Organizations (DAO).

2.3 Some Blockchain applications

Great attention, so much so that there is talk of a "gold rush", is paid to the cryptocurrency market: many investors, for fear of missing out (FOMO), rush to enter the market without full knowledge and awareness of the details and pitfalls. In fact, it is a market characterized by extreme volatility (the danger increases further when investing in coins recently introduced on the market or in the ICO phase), and by regulations that are not yet well-defined. This has contributed to the birth of scam projects. However, the cryptocurrency market is not the only investment opportunity. Possible applications of Blockchain technology have favored the development of many projects in different fields of the industrial world.

2.3.1 Financial Services

The application of Blockchain technology will certainly increase the efficiency of financial services such as payments, even if from a technological point of view, these are among the slowest services to adapt to new technologies. To transfer money, for example, from Italy to South Africa, it is necessary to go through numerous banks in a process that can take days or weeks, with costs that can reach more than 10% of the value of the transaction. Blockchain technology, by removing intermediaries, will be able to significantly streamline the process, reducing time and costs: Accenture has estimated that the savings linked to the use of Blockchain will be able to bring banks overall savings of over 3 billion dollars (out of a total expenditure of 30 billions of dollars)[5].

Of note is the partnership signed by companies such as Unicredit with *Ripple*. One of Ripple's products, *RippleNet*, is used to transform any asset (from fiat currencies to financial products) into tradable tokens within a distributed ledger [6]. In the field of banking, Blockchain is also inserted into the management of digital identity, establishing itself as a "single source of truth", certifying the authenticity of data and the immutability of information. By moving digital identity management to Blockchain, users are guaranteed the ability to create and have full control of their identity through which, once verified, they will be able to give access to their information to third parties.

2.3.2 Industry 4.0

The Blockchain technology can directly intervene in streamlining the *supply chain*, bringing transparency, product traceability and reducing the necessary trust between parties. A practical case is that of Carrefour which has begun to implement the IBM Blockchain for the traceability of its products: in concrete terms, on the label of each of Carrefour's products, there will be a QR code that consumers will be able to scan with their smartphone, and access information such as the name of the breeder, place and method of breeding, food administered, treatments, and other data.

Many other sectors are carrying out Blockchain-based projects: some examples of applications are related to the government and public administration fields, to the healthcare, to the automotive, and other different sectors, obviously together with the insurance services, such as the Use Case proposed in this Thesis work.

2.4 Analysis of the Thesis's Use Case

The Blockchain solution proposed in this Thesis work consists in a Smart Contract. For the purposes of this work, it is developed to define a Gamification mechanism: the data, sent to the Smart Contract by the Mobile App after the processes of data-detection and data-analysis, are used to determine a final ranking of the players. However, it can be considered as a starting point to exploit the Blockchain technology's potential. In a real context, considering this work as born to be developed together with Insurance Companies, a solution overcoming the limits of the traditional Personalized Car Insurances (PCIs) [2] could integrate a Smart Contract into the PCIs' design: it would allow to transparently define the driving style evaluation protocol and the calculation algorithms (with associated transparency and immutability of the processed data) that determine the auto-premiums underlying the PCIs.

2.4.1 Technology's Key Elements

There are basically two main trends/needs which can lead ICs to the integration of PCIs with Smart Contracts:

- the growing demand by drivers of *data/process transparency* and *data-privacy preservation*;
- an increased focus and interest on the *Blockchain technology*;

Starting from those trends, a qualitative analysis of the impact that the integration of such technology with the PCI's design could have on the drivers' perception, and on the ICs' offer as well, is given. It is reasonable to highlight the main elements of added value brought by the technology's adoption:

- *Higher level of data/process transparency*: by deploying insurance contracts on the Blockchain, ICs can support public verification of data collection/processing.
- *Greater trust in the system:* Smart Contracts are written in a programming language, so they are free of ambiguity and contain all the necessary logic within them: they define the rules and automatically require the parties involved to respect them, in a decentralized way, without the need to rely on central authorities. Smart Contracts can be seen as applications IFTTT (*If This Then That*): when the conditions of the contract are satisfied, the Smart Contract autonomously carries out specific actions [7].
- *Greater design flexibility:* the adoption of Smart Contracts and more generally the exploitation of the Blockchain technology makes real the possibility to think to an autonomous car (smart car) which identity is associated to the digital identity of a person, making easier some procedures, such as the transfer of ownership or the car's rental. It could also be possible, by relating to the PCIs, for a smart car to pay the insurance independently, carrying out microtransactions for every kilometer travelled. Or, as proposed with this solution, to integrate insurances with Smart Contracts providing instant transfers of funds whenever car accidentals happen, or harsh driving-events are detected.
- *Higher level of automation:* as embedded in a Smart Contract's properties, the IFTTT remove the need for the presence of central authorities or third parties.

2.4.2 Technological Paradigms

Starting from the previous considerations, it is possible to deduce how today, next to the traditional design process of the PCIs (shown in Figure 2.1), a new technological paradigm based on the Blockchain technology is emerging (shown in Figure 2.2).

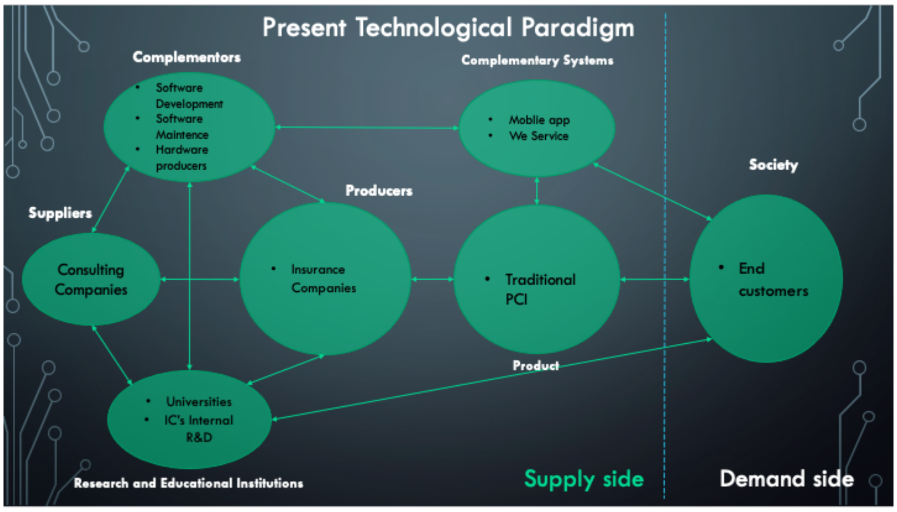


Figure 2.1: Present Technological Paradigm: Traditional PCI

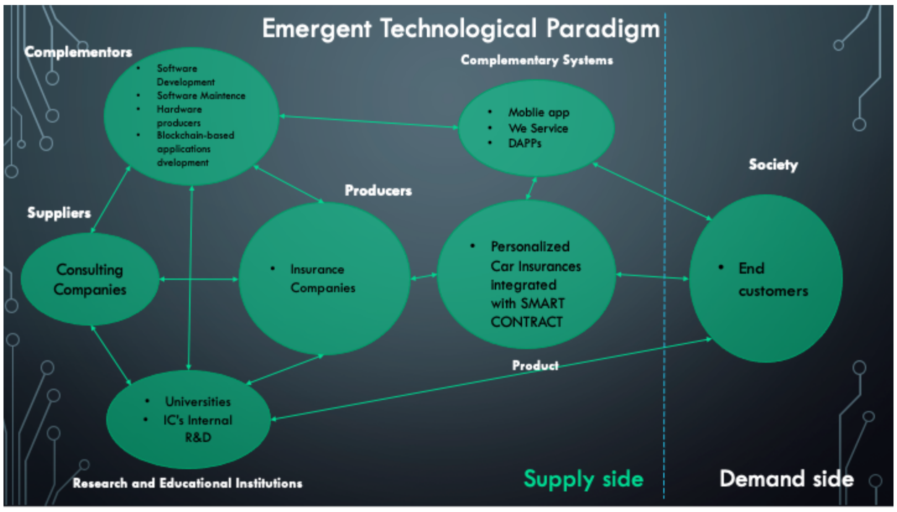


Figure 2.2: Emergent Technological Paradigm: PCI integrated with Smart Contract

Concerning the Supply side,

if the traditional paradigm considers a product based on the traditional design process for PCIs, the emergent paradigm is integrating PCIs with a Smart Contract. Regarding the complementors, the traditional paradigm considers as complementary to insurances, in a context where the aim is to monitor and evaluate the drivers' behavior, those products such as Mobile Apps and Web Services that could be adopted to detect data while the users are driving (as in the Thesis work). The emergent paradigm also takes into consideration the possibility to adopt Decentralized applications, which logic implementation is totally defined on a Blockchain platform.

Both paradigms mention as complementors those entities responsible for software development and maintenance, hardware's components development and production (devices and black-boxes for data-detection); the emergent paradigm also involves those entities interested in the development of the Blockchain technology's applications. Regarding the suppliers, by focusing on the PCIs design, the consulting companies working on the algorithms implemented in the entire detecting process of the drivers' behavior play a crucial role in both the technological paradigms. The Research and Educational institutions, such as IC's internal R&D and Universities, give a huge contribution as well.

Concerning the Demand Side, the main objective of this work and of this use-case is to design PCIs able to make drivers (clients of the ICs) more confident about the system.

2.4.3 Analysis of Paradigms through KPIs

As the implementation of the Blockchain technology through the development of Smart Contracts is presented as the emerging paradigm, a descriptive analysis supported by the analysis of some performance indicators is provided to further understand the reasons behind the progressive confirmation of this paradigm in the future scenarios.

Focusing on the general advent of the Blockchain technology and on all the possible applications of such technology, as illustrated in Section 2.3 , this open, public (depending on the application context) and secure system is rapidly becoming a strategic priority for many companies, with the premise of *reducing costs and inefficiencies*, radically transforming the business models known today. Furthermore, for the first time, thanks to the Blockchain, is possible *to solve the problem of "double-spending"* (related to the possibility of duplicating and therefore spending digital money) without the need of a central authority. However, the revolutionary aspect of this technology is not limited to this. It allows, as mentioned, to address one of the society's key aspects in a different way: *the problem of trust.* It has been solved by developing a technology in which trust is intrinsically within the technology itself: it is guaranteed through a combination of cryptography, consensus protocols and an economic system that encourages actors to cooperate with the rules defined by the Blockchain.

Moving the focus to the considered use-case, the following indicators have been chosen to analyze the performance of the Smart Contract when specific conditions are satisfied: direct transfer of funds in case of harsh events detected, direct transfer of funds at the end of a car travel according to the trip's evaluation score (as proposed in the Thesis work), or direct transfer of funds per kilometer travelled (insurance paid per kilometer travelled and not on monthly or annual basis).

1. *TPS: [tx/s]*

   the TPS, "transactions per second" indicates the number of transactions that a Blockchain network can handle in one second. It is an indicator of the speed and performance of the network, illustrating the ability to handle numerous transactions simultaneously.

2. *Average Execution time of a transaction: [min]*

   it refers to the time required for a transaction to be executed, and for a block to be added to the Blockchain. In this case, the time is referred to the transfer of funds when the above described conditions are met.

3. *Average Transaction fee: [wei, $]*

   the Average Transaction Fee measures the average fee when transaction is processed by a miner and confirmed. Focusing on the Ethereum Blockchain (the adopted Blockchain for this project), Ethereum Average Transaction Fee measures the average fee in USD when an Ethereum transaction is processed by a miner and confirmed.

4. *Average Cost of a transaction: [wei, $]*

   focusing on the Ehereum Blockchain (the adopted Blockchain for this project), the cost of a transaction is defined as follows.

   $$Cost_{Transaction} = Gas_{Limit} * Gas_{Price} \tag{2.3}$$

   The Gas Limit defines the maximum amount of gas that can be consumed in one transaction: in case the transaction requires a higher amount of gas to be executed, the transaction execution is interrupted when the gas limit is reached; in case the gas amount required by the transaction is lower than the defined gas limit, the exceeding gas is returned to the sender [3]. The Gas Price is the number of *wei* to be paid per unit of gas: if a transaction uses 10 gas and the gas price is set to 100, the total cost of the transaction is 1000 (10*100) *wei*.

Complex transactions require more gas, and therefore provide higher transaction fees.

In this case, the cost is referred to the transfer of funds when the above described conditions are satisfied.

By looking at some current data about the Ethereum Blockchain, some considerations follow:

- The average block generation time to confirm an Ether transaction is approximately 12 seconds compared to 10 minutes for Bitcoin [8].
- Focusing on the Ethereum Blockchain, the average fee for transactions has dropped on February 26, 2024, to 0.0015 ETH or $1.57, a figure previously seen in December 2020. Between January 2021 and May 2022, for almost two years the average commission requested by the network was around $40, recording its all-time high of $196,638 on May 1, 2022, as shown by BitInfoCharts data [9].

By considering the early stage of the Blockchain technology's development, it is important to specify that a lot of challenges have to be faced and are being faced: the most important one is related to the system's scalability. In a Blockchain, the system's scalability refers to the ability of handling an increasing number of transactions without affecting the system's performances. A limited scalability can be the reason behind low TPS, high average execution time of a transaction, high average transaction fee, and high average cost of a transaction.

Many solutions have been studied to improve the systems' scalability with consequent positive effects on the other indicators. The one proposed in this analysis, which is considered the main factor supporting the thesis that this emerging paradigm will definitely emerge, is the*Lightening Network*.

The Lightening Network is a payment system that allows to instantly send and receive payments, while reducing transaction costs. It is a second-level payment protocol built on the basis of a Blockchain, consisting of a network of two-way payment channels between users. The final result is to manage a series of transactions on the second level *off-chain* and to record only the final balance on the Blockchain later, through a single transaction when the channel is closed. It follows that a Blockchain implementing the Lightening Network protocol is able to carry out transactions instantly and at the same time drastically reduce the workload to which the Blockchain is subjected. The improvement of the system's scalability leads to the improvement of all the other mentioned performance indicators, which result to be very poor in case of system congestion.

In summary, by considering the crucial contribution of the Lightening Network to improve the scalability of a Blockchain system, it is possible to support the thesis according to which in the future scenarios the paradigm based on Blockchain applications will emerge. In this specific use case, this analysis highlights the importance of integrating the PCIs with a Smart Contract in order to satisfy the drivers' needs.

2.4.4 Radical and disruptive innovation

Starting from this Use-case, but considering the Blockchain technology as a whole and the specific properties of all the applications developed on it, it can be observed that is not a normal technological transition, but rather a total paradigm shift in the key characteristics of today's businesses: centralization becomes decentralization, closed systems become open, moving from being territorial to global entities, guaranteed by a trust no more placed in individual entities but in the very foundations of the technology. It is the case of a *radical* innovation with *disruptive* impact on the whole systems: in fact, even if the technology is still in the prototyping phase, with few players in every sector exploiting it, if one think to the Blockchain applications in the industrial sector, it can be assumed that it will soon become a determinant source of competitive advantage for companies, and a strong constraint for those incumbent firms not investing on the technology. If one think to the personal use as well, it cannot be ruled that it will become a technology totally integrated into the infrastructure of the applications used on a daily basis similar to how the Cloud is used today. In the next years there will not be a total substitution of the old paradigm, due to the many factors (regulation, risks associated with the adoption of a new technology, disinformation) but this emergent innovation will surely change the relationships between the main forces in the industry.

2.4.5 Technology's Development process

Even in this case, the focus is on the Blockchain technology as a whole. However, still supporting the potential of this technology, if is true that exists a technology removing the role of intermediaries and allowing the value exchange without central authorities, it is also true that, as common in the innovation pattern, the infrastructure needed to make the Blockchain proliferating is missing. To describe the current development phase of the Blockchain technology, at first an analogy between the Blockchain development process and the software development process is proposed; following this first part, an analysis of two s-curves is presented: the first s-curve representing the evolution of a performance indicator (daily transactions volumes) against the time; the second one representing the diffusion of the innovation in the market. These two contributions will lead to understand the current phase the Blockchain technology is living within its development process.

Analogy between software and Blockchain development process

The software development process consists in four phases [3]:

1. *Development phase:* this is the phase dedicated to the technical development of a software or platform.

2. *Alpha phase:* this is the phases dedicated to test the developed software or platform in order to check it works properly. Regarding the Blockchain, in this phase errors are checked and consolidated in order to guarantee safety. Once this phase is completed, the Blockchain is made public and becomes accessible to anyone.

3. *Beta phase:* some beta-testers are chosen to test the software solidity. Regarding a Blockchain, as it is made public, is not possible to select some users for the beta-tests: the beta-testers will be the users who decide to use a particular Blockchain, with the intent to also contribute to its development. This category can be better identified as 'early adopter': as explained in the next section 2.4.6, they usually already have some specific competencies and decide to contribute because they believe in the technology potential.

4. *Launch on the market:* the more a technology implementation is intuitive and easy, the more the technology will be successful.

   Regarding the Blockchain technology, today is not yet the time in which the average user does not need specific programming competences or knowledge to exploit the technology.

Starting from these considerations, it is possible to assume that the Blockchain technology is in the early stages of the Beta phase.

Analysis of Performance s-curve and Diffusion s-curve

Before going into the details of the specific s-curves proposed, it is worth to specify how s-curves can be analyzed: when progressing along the s-curve, it is common to define three main phases which are respectively termed *incubation* (during which both performance and diffusion still have to "take off"), *diffusion* (when performance and diffusion grow significantly) and *maturity* (when they approach saturation) [10].

The first image (2.4) has been realized in Microsoft Excel, starting from data retrieved online: it represents the progress in time of the daily transactions' volumes on the Ethereum platform, from 2018 to January 2024 [11]. By comparing that curve with the traditional s-curve of a performance indicator (as shown in in Figure "a" of 2.3), and focusing on the different phases, it can be confirmed that the

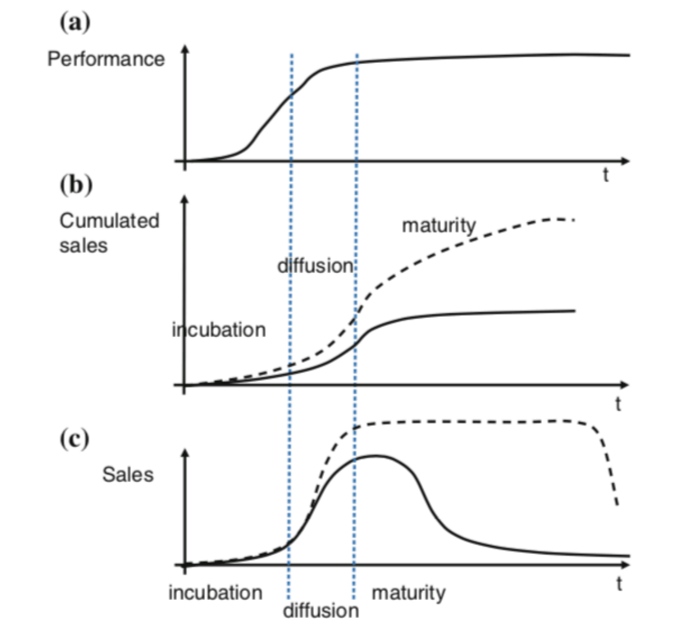


Figure 2.3: Performance and Diffusion s-curves.

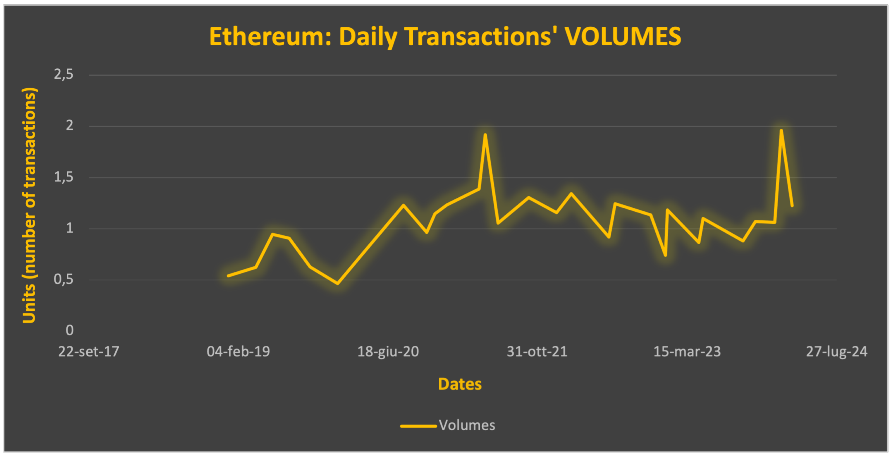


Figure 2.4: Ethereum: Daily Transactions' Volumes

Blockchain technology is going through the incubation period. The performance of the considered KPI has still to start to significantly grow. Conversely, it alternates some positive peaks to some significant negative peaks. It means the technology is still immature, even if the widespread awareness of the technology brings to promises associated to it that may be expressed in exaggerated terms. During the incubation period of a technological life-cycle, which is quite critical and interesting,

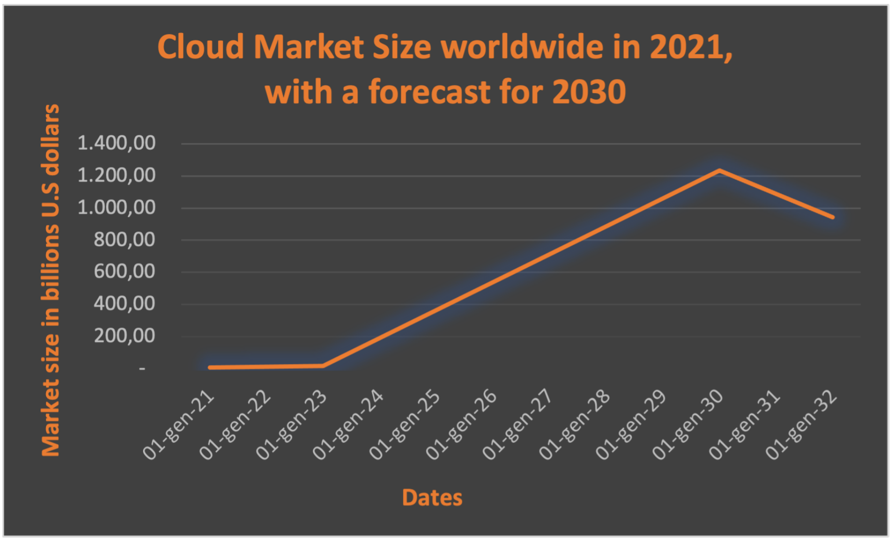


Figure 2.5: Cloud market size worldwide in 2021, with a forecast for 2030

it is in fact common to see technologies suffering from *hyper-inflated expectations*, or simply *hype*.

The incubation phase can be confirmed by the limited diffusion as well, as shown in Figure 2.5 [12]. The diffusion (or *penetration*) can be defined as the fraction of potential users that, at a given time, have decided to adopt the technology: in absolute terms, the diffusion curve represents the cumulated adoption sales, these being the sales to users who adopt the technology for the first time. Taking into consideration the bell-shaped curve (shown in Figure "c" of 2.3), and the adoption sales' curve (mathematically defines as the derivative of the diffusion s-curve), and imaging to build a s-curve over the histogram built on data related to Cloud market size worldwide in 2021 with a forecast for 2030 [12], it is possible to confirm that the Blockchain market size in this sector has still to take off.

From these two graphs, it can be confirmed the conclusion derived by the analogy of the Blockchain technology development process with the one of software: the current phase of Blockchain in the technology life-cycle is the incubation phase.

2.4.6 Diffusion of the innovation in the market

As above explained, due to the current state of the technology, to the lack of clear regulations and to the technology's intrinsic limits (scalability), the technology's adoption is far from the large-scale adoption. Referring to the most popular segmentation [10], shown in the Figure 2.6, the Roger's segmentation (based on the approximation to a normal curve of the diffusion sales curve) is proposed to determine the individual customers' segments who have approached the Blockchain

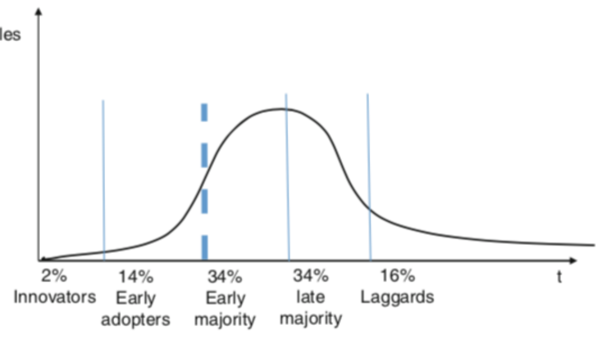


Figure 2.6: Market segments along the technological life-cycle

technology. Until today, according to the assumptions derived from the section 2.4.5, only the innovators and the early adopters have accessed this technology. The early-adopters proudly identify themselves as "visionaries", as they recognized the potential of a project or a product before others. Unlike users who approach with hesitation, they are not scared by any platform's problems, on the contrary they want to contribute to improve the platform's performance. Together with the early adopters, in the current phase of the technology's development process, also the contribution given by the hackers (they bring the platform toward higher security levels) and by the speculators (they exploit the cryptocurrency market's predisposition to speculation) must be considered. For the purposes of the platform development, speculators help to improve the functioning of the platforms themselves (i.e., to support high volumes of transactions), but together with hackers they increase the public perception that the cryptocurrency market is uncertain and full of pitfalls.

Anyway, the entry barriers to the cryptocurrency market and the entry barriers related to some technical aspects of the technology represent a natural status in a technology development and they are not stopping the spread of it or destroying its value. Every day new projects are born in every sector with the aim of exploiting this technology to develop innovative solutions and/or to reduce costs and inefficiencies: from the financial sector (Ripple project in the field of transaction and payments), to the supply chain (Carrefour's case); to the sector of anti-counterfeiting, smart home, automotive sector (self-driving car), insurance sector, government sector with projects related to digital identity and digital voting, and many others. For examples of Blockchain applications, see Section 2.3.

2.5 Gamification in the Insurance sector

Considering the revolution that the insurance world is experiencing, enabled by new business models and new technologies, it is interesting to consider, among the many innovations that will influence this industry, the role and impact of Gamification in the insurance sector.

In this regard, by focusing on the sector of car insurance policy, it is possible to cite the example of *Aviva* insurance.

Aviva, an English insurance company, provides an app to its car insureds that maps users' driving style by placing them in a competitive context. In case of download and use of the app for 200 miles, it gives the user a score out of 10 with 10 being the safest driver. This score then entitles the user to get money off his car insurance: the discount is personalized and calculated based on the individuals' driving behavior. The user will also receive feedback in the form of a star rating for his driving technique around accelerating, cornering and braking.

The game mechanism implemented in this Thesis work turns out to be very similar to those just proposed: in the developed Thesis product, the incentive does not reside in a reduction of the insurance's premium that the policyholder pays to the insurance company. Instead, it lies in the possibility of going to a prize pool in a Gamification mechanism in which the prizes are defined as percentages of a common fund, created by adding the fees paid by users when registering for the game, who, without registration, would not be able to access it. Wanting to bring the Thesis work closer to the insurance reality, on the basis of a final ranking, the drivers resulting as "the best three" could obtain a further percentage reduction in the premium, which would therefore depend not only on whether or not the insured caused accidents in the so-called "period of observation". The proposed Gamification mechanism could be integrated into the contractual formula of the *"Bonus-Malus"*: at each deadline, the progression or relegation of class will depend on whether or not claims have been caused, while as regard the progressive increases or percentage reductions of the premiums, the game mechanism may provide for the application of further discounts determined by a final ranking. In this way, driving style will be constantly monitored, drivers will be more stimulated to adopt a safe-driving style, and insurance companies will be able to benefit from it in terms of data collection and fewer accidents.

Apart from the structure of the Gamification mechanism, the crucial difference between the product proposed by *Aviva* and the one which is subject of this Thesis work lies in the *decentralization* concept: the Gamification mechanism proposed in this work is completely built on a Blockchain platform.

It is interesting to note how the Gamification mechanism has been tested in the insurance sector also in areas other than car policies.

- “PII Protectors” [13]: product created by *AllState*, used to train employees on privacy security and compliance. The game begins with a video showing the player being declined a mortgage because of identity theft. Thirsty for revenge, the player joins an agency to fight an evil conglomerate trying to steal other people’s data and chooses from four alter-egos: Captain Confidential, X-Ray Bex, Firewall and Raisa Sharp. Once players choose an identity, they are faced with dilemmas that they solve by answering questions on Allstate’s privacy policy. The more problems they solve, the more data their character stops from leaking outside of the company. This is a method to give *Allstate* employees a chance to learn the policy and experience how it translates to their day-to-day job while encouraging them to continue growing their understanding of its importance.

  Thus, in this case, the main objective of the Gamification mechanism consists in increasing the level of loyalty of its employees, and to increase the privacy of its data. Differently from the way the Gamification mechanism was exploited in the Thesis work, behind the *AllState* game there is not the Blockchain technology.

- “MyGame” [14]: is worth to mention the new features introduced in *Assicurazioni Generali*’s app, the well-known insurance group that operates mostly in Europe, North America and the Far East. It is a modest and rudimentary video game, little more than a quiz, the purpose of which, is purely didactic, little more than an expedient with which to prove the level of knowledge of the user on topics such as respect for the environment, health, knowledge of the regulations that regulate the circulation of cars and motorbikes. The cornerstone around which MyGame revolves are the 26 tests, divided into different categories. Each of them offers five multiple choice questions to choose from. Based on results, the user is rewarded both with an advancement in the statistics that make up his personal profile, a profile naturally included in a global ranking, and with a certain quantity of points that can be spent to progressively enhance certain decorative objects, such as a house, a backpack or a means of transport, an action that will certainly give user small satisfactions, but absolutely an end in itself.

  Even in the *Assicurazioni Generali*’s app, the Gamification mechanism is not built on a Blockchain platform.

- "Vitality Squares" [15]: launched in the life insurance sector by *Vitality Group*, the game provides educational information and also allows users to select and reveal what is behind a certain number of cards, which leads to a reward. The number of cards that can be turned over depends on the person's health status. For instance, if they have been engaging in healthy behaviors and are at platinum status, they can turn over six cards. And they can win a variety of prizes, including Starbucks and iTunes gift cards, and even a $500 Amazon gift card. So the members are motivated to take good care of their health so they can win more.

  From an ethical view point, this Gamification mechanism can be compared to the one designed in this work: the objective of the Thesis work was to incentivize users toward a safe driving-style as well as the *Vitality Group*'s game incentivizes users toward a safe healthy behaviour. However, even in this Case, the Gamification context is not built on a Blockchain platform.

Chapter 3

System Architecture

This work is based on the development of a Mobile Application that offers users the opportunity to interact with the Blockchain. More specifically, the project's goal is to monitor the driving behaviors of the application's users: with the aim of encouraging users to give their consent to being monitored, the evaluation process of the drivers' behaviors is inserted into a Gamification contest. The Gamification logic is defined by a Smart Contract, deployed on a Test-net.

3.1 Technologies

Before describing the system's architecture, an overview of all the explored technologies, in terms of software's libraries or development environments, is given:

- *Android Studio:* the Mobile app has been developed on Android Studio, which is the official integrated development environment (IDE) for Google's Android operating system, built on JetBrains' IntelliJ IDEA software and designed specifically for building Android apps [16]. The developed code is based on Java functions.

- *Remix:* it is an online editor allowing to develop Smart Contracts by using the Solidity programming language [17]. Thanks to the plugins, is possible to compile and test the code developed for the Smart Contract and, once the compiling process ends with success, the bytecode and ABI or the contract address can be used for distributing the Smart Contract.

- *Truffle*: it is a development environment, testing framework and resource pipeline for Blockchains using Ethereum Virtual Machine (EVM) [18]. It was used directly from command lines for generating the Solidity Javascript wrapper file.

- *Web3:* it is the software library used to interact with Smart Contracts [19]. Through the Web3j API it was possible to create the wallet addresses for the app's users, to read and write data from the Smart Contract, to create an instance of the deployed Smart Contract in order to call specific functions.
- *Infura:* it defines an end-point for accessing the Ethereum platform, and provides an API key for accessing the end-point and interacting with Smart Contracts [20].
- *Etherscan:* it is a block exploration and analysis platform on the one hand, and a decentralized Smart Contract platform on the other [21]. In this work, it was used for verifying the outcome of the transactions executed by calling the Smart Contract's functions.
- *Ethereum:* it is a Blockchain, open-source, decentralized, software platform [22]. In this work, the Sepolia Test-net is used, but the developed system is defined to work on Ethereum.

3.2 Architecture

The developed Mobile App puts itself in the middle between the End-Users and the Blockchain system. Looking at the Figure 3.1, we can distinguish:

- *End-User / Smartphone* (at the left of the Mobile App): the smartphone is the instrument used as specified in the following lines.
  - To detect driving-related data: with respect to the traditional black-boxes, the advantage of smartphones is that, thanks to the sensors they are already equipped with, smartphones can be used for detecting and collecting motion data without incurring in additional costs for hardware equipment. Furthermore, they have access to communication networks needed for data transfer and can process data onboard through the use of processors generally more powerful than the ones of black-boxes.
  - To allow the End-users, through the network connection, to access the Mobile App, and to directly interact with the Blockchain, in order to check the executed transactions, and to look at the data regarding the game saved into the Blockchain.
- *Blockchain* (at the right of the Mobile App):

  the Blockchain system is used to define the Gamification logic, to save and store the relevant data of the players and of the games by preserving the privacy and ensuring the transparency of data, and to manage funds directed

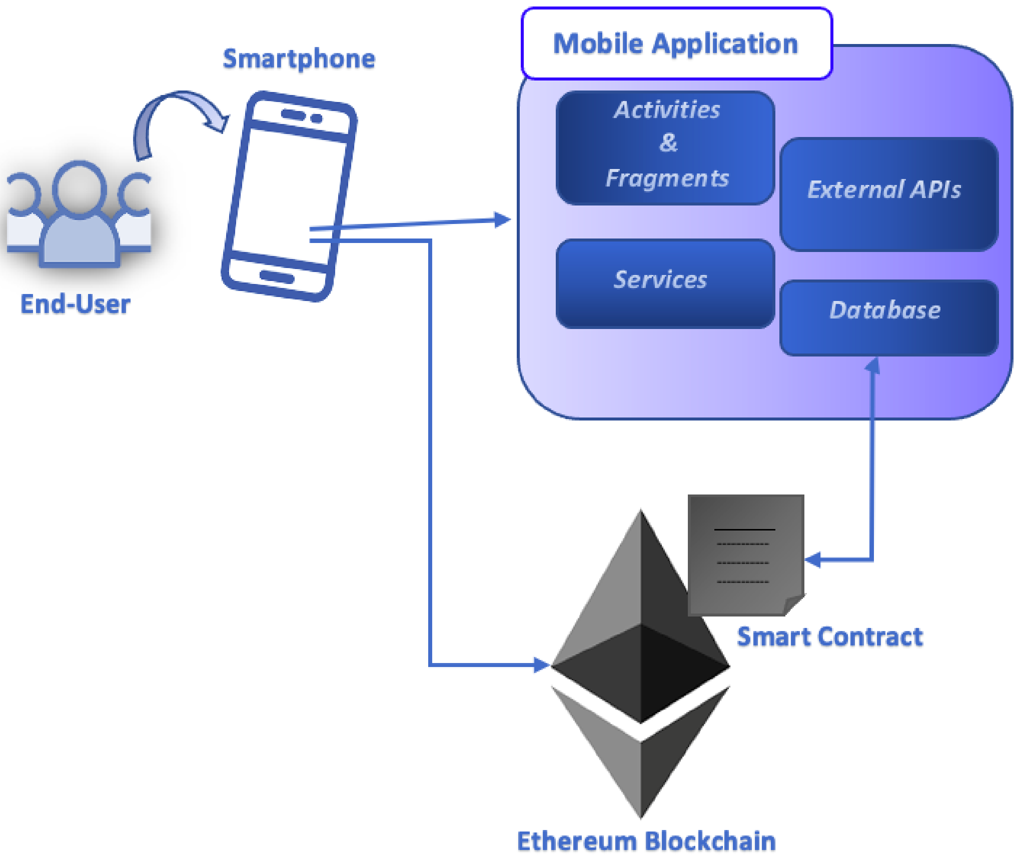


Figure 3.1: System Architecture

from the players to the Smart Contract, or from the Smart Contract to the players. Even if the system is defined to work on Ethereum, in this project's phase, is tested on Sepolia, a Blockchain test-net.

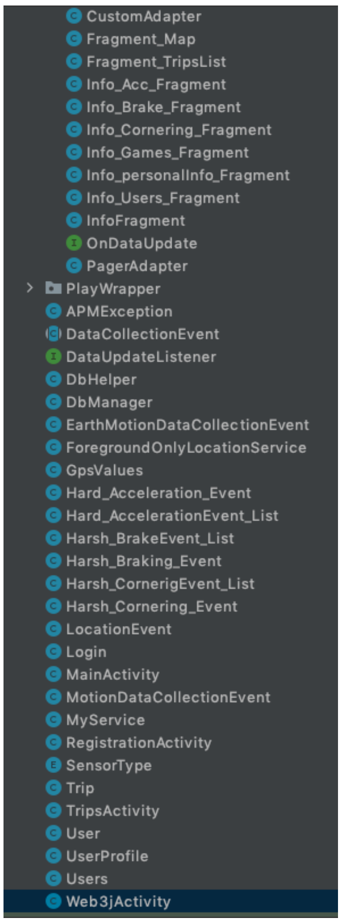
Focusing on the internal Architecture of the Mobile App, we can individuate different components:

- *Activities and Fragments*, defining the User interface (UI) and designed to make the user experience (UX) as easy as interesting: this section regarding the user experience will be further explored in Chapter **??**.
- *Services*, allowing to process data in background: services will be considered in the following Section 3.3, which describes the entire process of detecting the drivers' behaviors.

- *Database*, to store data related to the app's logic, and to retrieve data to be used for interacting with Blockchain: it will be defined in the section describing the application's data layer, which is part of Section 4.3.

- *External APIs*, for enabling the use of Google Services and the interaction with Blockchain: the section regarding the use of Google services is introduced in the following chapter in Section 3.4.5, whereas the interaction with Blockchain and the developed/deployed Smart Contract is in detail explored in Section 3.5.

All these components are punctuated by Java functions and represented by different java blocks or classes, as shown in Figure 3.2.



```
    CustomAdapter
    Fragment_Map
    Fragment_TripsList
    Info_Acc_Fragment
    Info_Brake_Fragment
    Info_Cornering_Fragment
    Info_Games_Fragment
    Info_personalInfo_Fragment
    Info_Users_Fragment
    InfoFragment
    OnDataUpdate
    PagerAdapter
> PlayWrapper
  APMException
  DataCollectionEvent
  DataUpdateListener
  DbHelper
  DbManager
  EarthMotionDataCollectionEvent
  ForegroundOnlyLocationService
  GpsValues
  Hard_Acceleration_Event
  Hard_AccelerationEvent_List
  Harsh_BrakeEvent_List
  Harsh_Braking_Event
  Harsh_CornerigEvent_List
  Harsh_Cornering_Event
  LocationEvent
  Login
  MainActivity
  MotionDataCollectionEvent
  MyService
  RegistrationActivity
  SensorType
  Trip
  TripsActivity
  User
  UserProfile
  Users
  Web3jActivity
```

Figure 3.2: Java Classes

3.3 Process of detecting the drivers' behavior

As shown in Figure 3.3, the process of detecting a driver's behavior usually consists in three main phases:

1. sensors' raw data acquisition and data transformation;
2. classification of driving events;
3. computation of a score for the driver, according to the evaluated driving data.

In the next sections, the work performed is going to be explained.

3.3.1 Sensors' raw data acquisition and data transformation

Data acquisition: Hardware component

Starting from existing literature [1], one of the assumptions derived recognizes and supports the ability of smartphones in recording and processing driving-related data. In this case, the data acquisition is performed by the sensors the smartphone with which it is already equipped: it is worth to specify that accelerometer readings are detected in the phase of data acquisition, then pre-processed trough the gravity readings detected by the gravity sensor in the phase of data transformation, whereas the rotation sensor's readings are acquired and considered for determining the device's orientation, as explained in Section 3.4.6.

The accelerometer is a motion sensor: it measures the acceleration force in in m/s^2 that is applied to a device on all three physical axes (x, y, and z), including the force of gravity. It uses the standard sensor coordinate system, knowing that the coordinate system is defined relative to the device's screen when the device is held in its default orientation (see Figure 3.4). In practice, it means that the following conditions apply when the device is held in its default orientation:

- the x axis is horizontal and points to the right;
- the y axis is vertical and points up;
- the z axis points toward the outside of the screen face; in this system, coordinates behind the screen have negative z values.

It's worth to specify that, even when the device moves and the device's screen changes orientation, the axes are not swapped, so the sensor's coordinate system never changes.

Going deeply into the raw data acquisition by the sensor, by considering the axes orientation in the device [23], the following data are detected by the accelerometer:

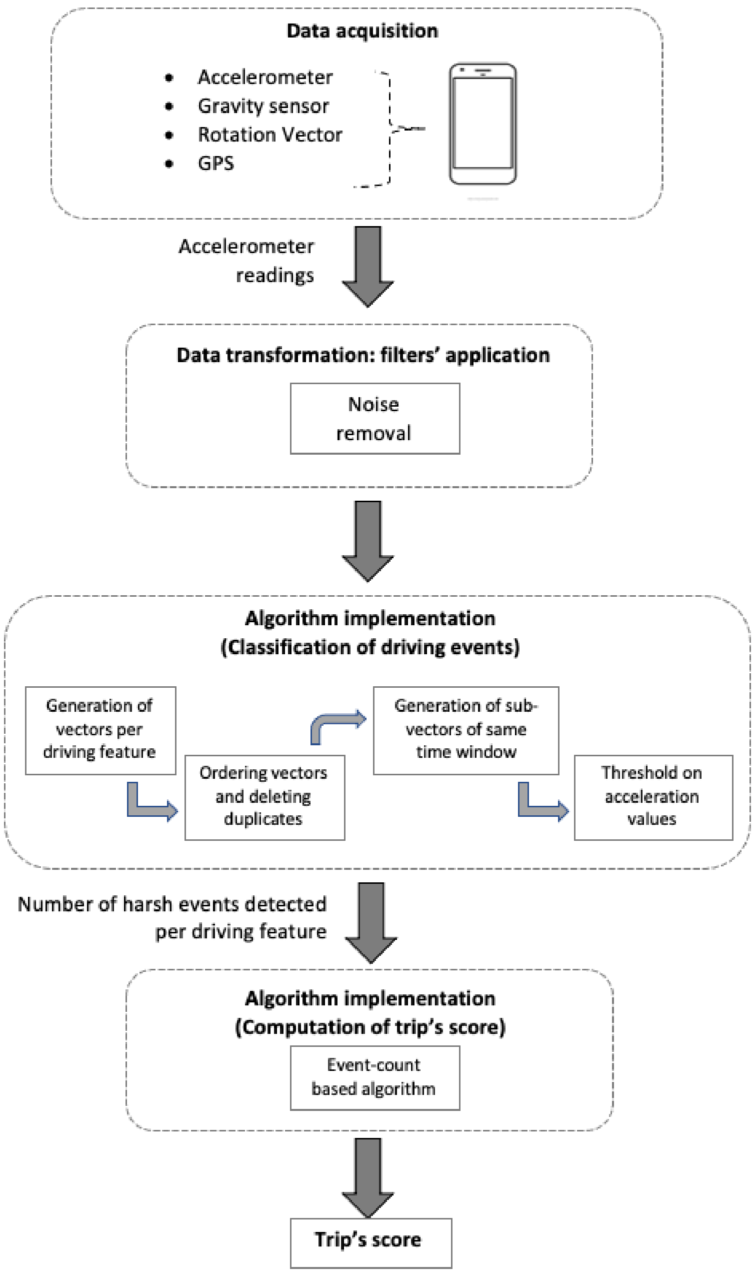

Data acquisition
Accelerometer
Gravity sensor
Rotation Vector
GPS
Accelerometer readings
Data transformation: filters' application
Noise removal
Algorithm implementation (Classification of driving events)
Generation of vectors per driving feature
Ordering vectors and deleting duplicates
Generation of sub-vectors of same time window
Threshold on acceleration values
Number of harsh events detected per driving feature
Algorithm implementation (Computation of trip's score)
Event-count based algorithm
Trip's score

Figure 3.3: Process of detecting the drivers' behavior

- readings on the z axis, for frontal acceleration;

- readings on the x axis, for lateral acceleration.

As the vehicle is moving frontally and laterally and the device is kept in a stationary position, with the screen in in its default orientation, readings on the y axis are not considered: the stationary device will have an acceleration value detected on the y axis of $+9.81$ m/s^2, which corresponds to the acceleration of the device (0 m/s2 minus the force of gravity, which is - 9.81 m/s^2).

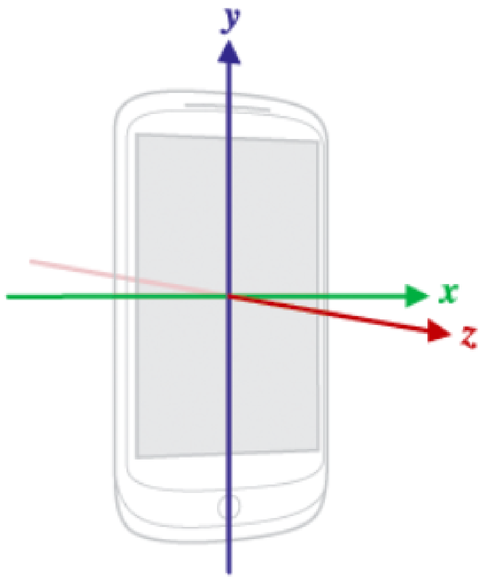


Figure 3.4: Sensor's coordinate system. About the device's orientation, it is important to remark that, even if the device's orientation is changing, the disposition of the axes remains the same. Then, the same readings can be detected on the same axes, independently on whether the smartphone is kept horizontally or vertically.

Data acquisition: Service

Moving the focus away from the hardware component, in order to collect the driving data of the user during each trip, a service called *MyService* has been implemented.

A Service is an application component that can perform long-running operations in the background. It does not provide a user interface. Once started, a service might continue running for some time, even after the user switches to another application.

The connection with the service starts when the *startTrip* method in the *TripsActivity* component calls the *startConnectionWithService* method, which in turn includes the *startService(Intent intent)* method. The intent specifies the class of the Service to which the call is directed. The role of the Service is to get the available sensors of the device, to register the listeners on the available sensors and to save the data of each new event on the app's database, in order for the *TripsActivity* component to retrieve and use them.

The Service can access the device's sensors and register the listeners on them through the *SensorManager* class. By implementing the *SensorEventListener* interface, after registering the listeners, the Service starts collecting sensors' data in the form of event, every time new events are available (values' changes are detected by sensors on the axes). This object of the *SensorEvent* Class holds information such as the sensor's type, the timestamp of the event, accuracy and the values detected on the sensor's axes. Each detected event in the *OnsensorChanged* method is executed one-by-one by an *AsyncTasks*: in the *doInBackground* method the event's required parameters are collected, pre-processed to remove gravity influence and the noise, and inserted in the database. In this case, the values derived from the ones detected by the accelerometer on the z axis are saved into the Table *SensorData_zValues*, together with the timestamp of the event. Whereas the values derived from the absolute values detected by the accelerometer on the x axis are inserted into the Table *SensorData_xValues*, together with the timestamp. As shown below, in the code related to the *doInBackground* method, the values registered on the y axis are not saved into the database.

The connection with the Service ends with the *stopTrip* method in the *TripsActivity* component, where the *stopConnectionWithService* method is called. This method contains the *stopService(Intent intent)* command, which recalls the *onDestroy* callback method in the Service: here, through the *SensorManager* class, the Service unregisters the sensor's listeners. Also in this case, the intent specifies the Service class to which the call is directed.

3.3.2 Data transformation

In order to define the final dataset of values to be saved into the database and to be analyzed with the aim of classifying driving events, the raw data (*SensorEvent* data) will be pre-processed by using *low-pass* and *high-pass* filters to eliminate gravitational forces and reduce noise [24]. The developed code uses a simple filter *constant (alpha)* to create a low-pass filter. This filter constant is derived from a time constant (t), which is a rough representation of the latency that the filter adds to the sensor events, and the sensor's event delivery rate (dt). As the purpose is to get sensor's data as fast as possible (*dt=0*), the alpha value adopted is equal to 1.

In the code below, the implementation of low-pass and high-pass filters, inserted into the *doInBackground* method. As shown, the values detected by the gravity sensor on the three axes are used to apply the low-pass filter first, and to apply the high-pass filter later. The gravity sensor provides a three-dimensional vector indicating the direction and magnitude of gravity. Typically, this sensor is used to determine the device's relative orientation in space. In this case, it used to optimize the raw data derived from the accelerometer, affected by the gravity's influence and noises. At this time, data are ready to be organized in the app's database, and to be analyzed.

```
@Override
   public void onSensorChanged(SensorEvent event) {
       new SensorEventLoggerTask().execute(event);
   }
   @SuppressLint("StaticFieldLeak")
   private class SensorEventLoggerTask extends AsyncTask<
  SensorEvent, Void, Void> {
       @Override
       protected Void doInBackground(SensorEvent... sensorEvents)
   {

       // Collecting event values
           SensorEvent event = sensorEvents[0];

           switch (event.sensor.getType())
           {
           // Gravity sensor
               case (Sensor.TYPE_GRAVITY):
                   gravity_x = event.values[0];
                   gravity_y = event.values[1];
                   gravity_z = event.values[2];
                   break;

           // Accelerometer sensor
               case (Sensor.TYPE_ACCELEROMETER):
                   final SensorType st = SensorType.ACCELEROMETER
  ;
                   MotionDataCollectionEvent dataCollectionEvent
  =
                           (MotionDataCollectionEvent) st.
  createDataCollectionEvent(event,
                           new Date(System.currentTimeMillis()));
                   x_component_acc = event.values[0];
                   y_component_acc = event.values[1];
                   z_component_acc = event.values[2];
                   timestamp_event = dataCollectionEvent.
  getTimestamp();
                   event_timestamp = timestamp_event.getTime();
                   break;

           // Rotation vector
               case (Sensor.TYPE_ROTATION_VECTOR):
                   rotation_matrix = new float[16];
                   SensorManager.getRotationMatrixFromVector(
  rotation_matrix, event.values);
                   defineOrientation(rotation_matrix);
                   break;
           }
            //low-pass filter
           final float alpha = 1;
```

```
            gravity_x = alpha * gravity_x + (1 - alpha) *
    x_component_acc;
            gravity_y = alpha * gravity_y + (1 - alpha) *
    y_component_acc;
            gravity_z = alpha * gravity_z + (1 - alpha) *
    z_component_acc;
            Log.i(TAG, "low-pass filter applied!");

            //high-pass filter
            double linear_acceleration_x = x_component_acc -
    gravity_x;
            double linear_acceleration_y = y_component_acc -
    gravity_y;
            double linear_acceleration_z = z_component_acc -
    gravity_z;
            Log.i(TAG, "high-pass filter applied!");

            //save data in DB
            insertZvalues(db, linear_acceleration_z,
    event_timestamp, newSensorDataId_Z() );
            insertXvalues(db, Math.abs(linear_acceleration_x),
    event_timestamp, newSensorDataId_X() );
        }
```

Listing 3.1: doInBackground() method

3.3.3 Classification of driving events

Before describing the developed code for determining an event as "harsh event", it is worth to introduce some considerations on the adopted algorithm. In the literature, three main approaches have been explored to classify driving events: anomaly detection, threshold, and machine learning classifier-based methods. As described in the article [1], the different methods have been applied to the publicly datasets available at the time the article was written (Ferreira's Dataset and Carlo's Dataset), and on an additional dataset "ad hoc" created and based on driving data recorded through multiple devices, one of which was an Android Smartphone (AD2 Dataset). All the algorithms were performed by changing the reference values (threshold and length of time window), in order to find those values optimizing the performance.

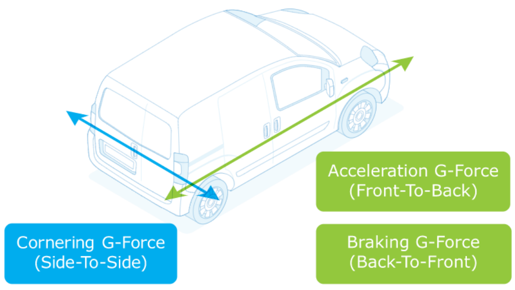
The algorithms' performance has been evaluated (by computing precision, recall and F-score) according to two different set of experiments: the first experimental setup focused on the classification of labeled events, then the classification performance of each algorithm was evaluated on labeled aggressive vs. non-aggressive events; the second setup included all available acceleration data, also those recorded during normal driving situations, so the detection performance of each algorithm was evaluated on the entire dataset.

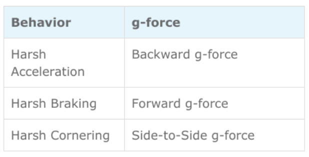
By relying on the results achieved through the work performed for the article [1] (as shown in Figure 3.6), and referring to results achieved by the different algorithms when performed on the entire dataset, an this work, one of the threshold-based methods has been adopted. The decision to focus on all the available acceleration data detected during normal driving situations depends on the consideration of this approach as more closely reflecting a realistic scenario, where the task performed is more a detection task rather than a classification task. The threshold-based method adopted for detection of events is, as defined in the article, the *"simple-threshold"* method.

According to this algorithm, by referring to the individual sensor's readings, as shown by the green line and blue line in Figure 3.5:

- A sensor reading on the z axis (related to frontal acceleration) is labeled as "aggressive" when the value is higher than a certain acceleration threshold, or lower than a certain brake threshold.

- A sensor reading on the x axis (related to lateral acceleration) is labeled as "aggressive" when the absolute value of the acceleration is higher than a certain turn threshold.


Acceleration G-Force (Front-To-Back)
Cornering G-Force (Side-To-Side)
Braking G-Force (Back-To-Front)



Behavior	g-force
Harsh Acceleration	Backward g-force
Harsh Braking	Forward g-force
Harsh Cornering	Side-to-Side g-force

Figure 3.5: Orientation of sensor's coordinate system in vehicles

By referring to a driving event, the detection of the "harsh event" also depends on the time window considered for the event. Different events may need different time windows: i.e. u-turn event takes longer time than a frontal acceleration.

A driving event is defined as "aggressive" if at least 50% of the time points belonging to the window is labeled as aggressive (if at least 50% of sensor's readings exceeds the considered threshold), otherwise as "non-aggressive".

Moving to the work performed in this project, the values adopted for the thresholds and the time window, as shown in the table in Figure 5, are the ones corresponding to the "simple-threshold" method, applied on AD2 Dataset, with focus on all the available acceleration data:

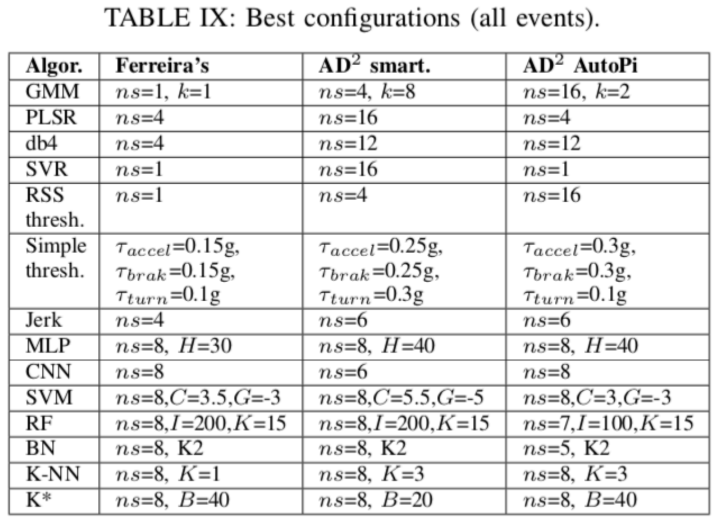


TABLE IX: Best configurations (all events).

Algor.	**Ferreira's**	**AD^2 smart.**	**AD^2 AutoPi**
GMM	ns=1, k=1	ns=4, k=8	ns=16, k=2
PLSR	ns=4	ns=16	ns=4
db4	ns=4	ns=12	ns=12
SVR	ns=1	ns=16	ns=1
RSS thresh.	ns=1	ns=4	ns=16
Simple thresh.	τ_{accel}=0.15g, τ_{brak}=0.15g, τ_{turn}=0.1g	τ_{accel}=0.25g, τ_{brak}=0.25g, τ_{turn}=0.3g	τ_{accel}=0.3g, τ_{brak}=0.3g, τ_{turn}=0.1g
Jerk	ns=4	ns=6	ns=6
MLP	ns=8, H=30	ns=8, H=40	ns=8, H=40
CNN	ns=8	ns=6	ns=8
SVM	ns=8,C=3.5,G=-3	ns=8,C=5.5,G=-5	ns=8,C=3,G=-3
RF	ns=8,I=200,K=15	ns=8,I=200,K=15	ns=7,I=100,K=15
BN	ns=8, K2	ns=8, K2	ns=5, K2
K-NN	ns=8, K=1	ns=8, K=3	ns=8, K=3
K*	ns=8, B=40	ns=8, B=20	ns=8, B=40

Figure 3.6: Table with reference thresholds

- *acceleration* threshold: 0.25g (0.25 * 9.81 m/s^2 = 2.45 m/s^2);
- *brake* threshold: 0.25g;
- *turn* threshold: 0.3g;
- *ns* = 4 seconds.

These thresholds define the exception rules: data exceeding these values define exception values and when the number of exception values is higher than 50% of the values considered within vectors of time window equal to 4 seconds, a harsh event is detected. From here on, the work performed for the detection of harsh events is in detail illustrated. According to the already mentioned article, the classification of driving events can be either binary (aggressive or non-aggressive event), or multi-class, i.e., to identify whether the event was related a (an aggressive) specific driving maneuver. In this work, the *SensorEvent* data collected during each trip will be organized and analyzed so that, starting from detection of eventual aggressive driving events, also the related harsh event is individuated.

- Which harsh events?

- *Harsh acceleration*, by analyzing the recorded events containing positive values detected by the accelerometer on the *z* axis.

- *Harsh Brake*, by analyzing the recorded events containing negative values detected by the accelerometer on the z axis.

- *Harsh Cornering*, by analyzing the recorded events containing the absolute values detected by the accelerometer on the x axis.

In order to get those values, a SELECT query with a WHERE clause is executed on the two tables above mentioned (SensorData_xValues, SensorData_zValues). It will return a cursor pointing only to those rows of the tables having the event's *timestamp* included between the *start date* and the *end date* of the trip. If the cursors are not empty:

1. The first step of the data analysis leads to the creation of the following arraylists of events.

   - *Hard_acceleration_events* arraylist: it's a list of events,

     from the *Hard_Acceleration_Event* Class. Each item of the list is a hard acceleration event, containing the positive value detected on the z axis and the event's timestamp.

   - *Harsh_braking_events* arraylist: it's a list of events,

     from the *Harsh_Braking_Event* Class. Each item of the list is a harsh braking event, containing the negative value detected on the z axis and the event's timestamp.

   - *Harsh_cornering_events* arraylist: it's a list of events,

     from the *Harsh_Cornering_Event* Class. Each item of the list is a harsh cornering event, containing the absolute value detected on the x axis and the event's timestamp.

   The developed code is illustrated for the *hard_acceleration_events* arraylist below.

```
  private ArrayList<Hard_Acceleration_Event>
create_HardAcc_event(double _zcomponent, long _timestamp) {
  Date timestamp_event = new Date(_timestamp);
  Hard_Acceleration_Event new_event = new
Hard_Acceleration_Event(timestamp_event, _zcomponent);
  hard_acceleration_events.add(new_evento_AccBrusca);
  return hard_acceleration_events;  }
```

Listing 3.2: create hard acceleration events arraylist

   If the size of the defined arraylists is different from zero:

2. The second step of the analysis consists in ordering the lists according to an increasing value of the timestamp, and in deleting possible duplicates of the events from the lists. The developed code is illustrated for the *hard_acceleration_events* arraylist, in the following lines.

```java
    private void ordering_deletingDuplicates(ArrayList<Hard_Acceleration_Event> hard_acceleration_events) {

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.N) {
        hard_acceleration_events.sort((hard_acceleration_event, t1) -> hard_acceleration_event.getTimestamp().compareTo(t1.getTimestamp()));
    }

    for (int i = 0; i < hard_acceleration_events.size() - 1; i++) {
        if (hard_acceleration_events.get(i).equals(hard_acceleration_events.get(i + 1))) {
            hard_acceleration_events.remove(i);
            i--;
        }
    }
}

```

Listing 3.3: Ordering arraylists and deleting events' duplicates

3. The third step of the analysis consists in constructing sub-lists of the mentioned lists, in order to analyze one-by-one vectors of events having a time window equal to four seconds (*ns=4*). Referring to the code, these sub-lists of harsh events are called:

   - *_container_vettori_hardAccEvent*, which includes the sublists of equal time window, derived from the initial *hard_acceleration_events* arraylist;
   - *_container_vettori_harshBrakingEvent*, which includes the sublists of equal time window, derived from the initial *harsh_braking_events* arraylist;
   - *_container_vettori_harshCorneringEvent*, which includes the sublists of equal time window, derived from the initial *harsh_cornering_events* arraylist.

   The developed code is illustrated for the *_container_vettori_hardAccEvent* in the following lines.

```java
    private ArrayList<ArrayList<Hard_Acceleration_Event>> create_Vectors_SameWindow_Acc(ArrayList<Hard_Acceleration_Event> _harsh_acc_events) {
```

```
    int ns = 4;
    int l = 1;
    int i;

    while (l < _harsh_acc_events.size()) {
        ArrayList<Hard_Acceleration_Event> list = new
ArrayList<Hard_Acceleration_Event>();

        for (i = l; l < _harsh_acc_events.size(); ) {
            Hard_Acceleration_Event evento = _harsh_acc_events
.get(i);
            long first_timestamp = _harsh_acc_events.get(i).
getTimestamp().getTime();
            long second_timestamp = _harsh_acc_events.get(i -
1).getTimestamp().getTime();

            if ((first_timestamp - second_timestamp) > ns *
1000) {
                l = _harsh_acc_events.indexOf(
_harsh_acc_events.get(i));
                l += 1;
                break;
            }
            list.add(evento);
            i++;
            l++;
        }
        container_Vectors_HardAccelerationEvent.add(list);
    }
    return container_Vectors_HardAccelerationEvent; }

```

Listing 3.4: Building up sub-lists of four seconds time window

4. At this point, the analysis process has come to the last step: the detection of harsh events. By scrolling one-by-one the events of the created sub-lists with time window of four seconds, if the number of exception values (values exceeding a given threshold in case of acceleration threshold and turn threshold or lower than a given threshold in case of brake threshold) is higher than half the size of the considered sub-list, a harsh event is generated. When moving to the other sub-lists containing the same type of harsh events, every time it occurs, the number of such harsh events increases by one. At the end of this step, the number of harsh events for the trip is determined and recorded in the following variables: *n_HardAccelerations_Trip*, *n_HarshBrakes_Trip*, *n_HarshCornerings_Trip*. Here, the developed code is illustrated for *n_HardAccelerations_Trip*.

```java
 private int detecting_HarshManoveurs_Acc(ArrayList<ArrayList<Hard_Acceleration_Event>> _container_vectors_hardAccEvent) {
 int n_HardAccelerations_Trip = 0;
 int i;

 for (int j = 0; j <= (_container_vectors_hardAccEvent.size() - 1); ) {
     ArrayList<Hard_Acceleration_Event> array = _container_vectors_hardAccEvent.get(j);
     int n = array.size();
     int hard_acc_values_singleArray = 0;

     for (i = 0; i < array.size() - 1; ) {
         double z_value = array.get(i).getValue();

         if (z_value < 2.45) {
          continue;
         }
         hard_acc_values_singleArray++;
         i++;
     }

     if (hard_acc_values_singleArray >= n / 2) {
         n_HardAccelerations_Trip++;
     }

     if (n_HardAccelerations_Trip > 0) {
         Cursor crs = db.getID_LocationHarshEvent(array.get(0).getTimestamp());
         long id_GpsData = 0;

         if (crs != null && crs.moveToFirst() && !crs.isNull(crs.getColumnIndex("_id"))) {
             do {
                 id_GpsData = crs.getLong(crs.getColumnIndex("_id"));
             } while (crs.moveToNext());
         }
         long id_SensorData = 0;
         double value_eventZero = array.get(0).getValue();
         long timestamp_eventZero = array.get(0).getTimestamp().getTime();
         Cursor crs_eventZero = db.getID_EventZero_Z(value_eventZero, timestamp_eventZero );

         if (crs_eventZero != null && crs_eventZero.moveToFirst() && !crs_eventZero.isNull(crs_eventZero.getColumnIndex("_id"))) {
```

```
                    do {
                        id_SensorData = crs_eventZero.getLong(
    crs_eventZero.getColumnIndex("_id"));
                    } while (crs_eventZero.moveToNext());
                }
                inserisciAccValues(db, id_SensorData , id_GpsData,
     currentUser.getUsername(), currentTrip.getId(),
    newHardAccId());
            }
            j++;
        }
        return  n_HardAccelerations_Trip;
}

```

Listing 3.5: Detection of harsh events

In Figure 3.7, there is a representation of a non-aggressive u-turn event (first event) and an aggressive u-turn event (second event), taken from the article. The red lines define the moments at which the driving event starts and finishes. In the first case, the number of values detected on the x axes that exceed the threshold is lower than 50% of the values defining the driving event. Conversely, in the second case, the number of exceeding values is higher than 50% of the total values detected for the driving event.

3.3.4 Score's Computation Process

Computation of the driver's score for the individual trip

The last phase of the process consists in computing the driver's score for the trip. As above said, the last step of the analysis process leads to the definition of the number of harsh events per trip. Then, it comes out with a value assigned to the following variables: *n_HardAccelerations_Trip*, *n_HarshBrakes_Trip*, *n_HarshCornerings_Trip*, which will passed in input to the *getYourTripScore* method. The *getYourTripScore* method, that will be called every time the user finishes a trip, implements the following algorithm:

1. Computation of a score for each driving feature (harsh acceleration, harsh brake, harsh cornering): this score depends on the number of harsh events per driving feature and on the number of trip's kilometers actually traveled. In case of trips with the same number of harsh events, the shorter the trip (the lower the trip's number of kilometers) the more severe is evaluated the impact of the harsh events; i.e. two events of harsh acceleration detected on a trip of 100 kilometers have a different impact on the trip's score than two events of harsh acceleration detected on a trip of 15 kilometers. In the former case

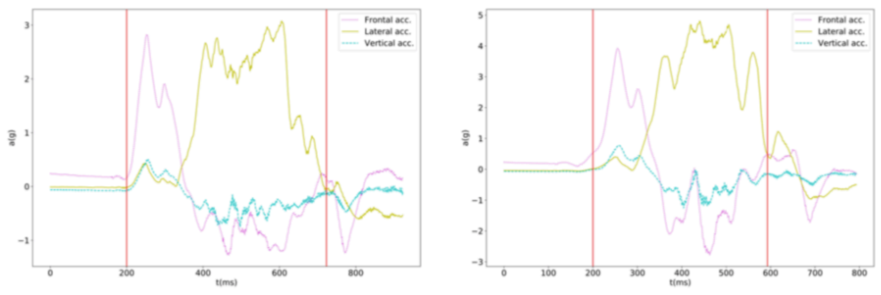


Figure 3.7: Harsh u-turn event

the score for the acceleration's driving feature of the trip is higher than the one computed in the second case. This is the formula used for computing the score for each of the driving features (for each trip):

$$harh\_event\_score = Max[max\_trip\_score^{\smile} \frac{harsh\_events\_number \cdot \beta}{trip\_km\_number}, 0] \tag{3.1}$$

where *max_trip_score* is a parameter set to 100, by hypothesizing a maximum number of user's trips equal to 10, and a maximum score reachable in the whole game by each user equal to 1000 (the *max_trip_score* is equal to the max score reachable in the game divided by the max number of user's trips); *harsh_events_number* contains the number of harsh events detected per each harsh event type (*n_HardAccelerations_Trip*, *n_HarshBrakes_Trip*,

n_HarshCornerings_Trip'); *beta* is the "conversion factor" introduced to distribute the number of harsh events on the trip's number of kilometers, with the aim of differentiating the impact of the number of harsh events according to the trip's length. As the trip's length is measured in kilometers, the *beta* value is 1000. The *Math.Max(,)* method is used because in case of bad driving behavior, the score for the individual driving feature could be equal to a negative vale: in that case the value for the score is zero.

2. Definition of a weight for each of the driving features considered: by considering that the sum of the weights should be equal to 100%, in this case the same weight was assigned to each of the driving features, with a value equal to 33,3% each.

3. The final score for the trip is defined as a weighted sum of the scores assigned to each of the driving features:

$$final\_trip\_score = \sum_{n=1}^{N} harsh\_event\_score_n \cdot harsh\_event\_weight_n \tag{3.2}$$

with $1 < n < 3$, as the current system considers three types of harsh events (acceleration, braking, and cornering events). The *harsh_event_weight* is used to increase or reduce the contribution of a specific harsh events type (since we decided to assign the same weight to each harsh event type, we set the weight equal to 33.3% for each n). It is worth remarking that the current equations currently consider all the harsh driving events as equals, without penalizing the score gained by the user based on "how harsh" the generated events were. In the future, the equations could be modified in order to take into account also how much the acceleration values recorded surpassed the threshold.

In summary, as the formulas show, this scoring method is *Event-count* based, according to the exception rules defined over the driving features. It's worth to notice that the scoring is calibrated to award a score of 0 if a driver incurs 10 events or more, over 100 kilometers driven or fewer.

Computation of the driver's score at the end of the Game

The final score of the driver is obtained by summing all the scores related to each trip, at the end of the 10 trips.

As explained, the last step of the analysis comes out with the detection of eventual harsh events.

For each harsh event determined, the location where the abrupt driving maneuver is committed by the user is defined. It's obtained by executing a SELECT query with WHERE clause on the table collecting location data from GPS: it returns a cursor pointing to those rows having the timestamp equal to the one of the "*zero*" harsh event. The zero harsh event corresponds to the first item of each sub-list that leads to the determination of a harsh event.

It allows the user to be aware of the relevant data characterizing each trip: number of harsh events, localization of harsh events. It also supports the transparency of the algorithm used for determining the trip's individual score. Concerning the detection of the user's location, and the update of the current location, the next section is going to explain the implemented *Location service* as both foreground Service and binding Service.

3.4 Service for Location Update

In order to request location updates when the app is visible and when the user navigates away from the app's activity, the component *TripsActivity* is bounded to a *Location service.*

A *bound* service is an implementation of the Service class that lets other applications or other applications' components bind to it, send requests, receive responses, interact with it [25]. In this case, it will provide constant location updates to the *TripsActivity* component [26].

This Location service subscribes and unsubscribes to location changes, and promotes itself to a foreground service (with a notification) if the user stops interacting with the app.

A *foreground* service performs operations that are noticeable to the user: when the app is working in the background, through a status bar notification, the service makes users aware that the app is performing a task in the foreground and is consuming system resources [27]. In this case, even when users navigate away from the app, they are notified about their current location.

3.4.1 Location Service as Binding Service

Going into the programming details, in order to provide binding for the service, the *onBind* callback method is implemented in the service *ForegroundOnlyLocationService.* This method returns an *IBinder* object defining the programming interface that clients can use to interact with the service. In this case, as only the local application is using the service and it does not need to work across processes, the programming interface has been defined by extending the *Binder* Class. The instance of this *Binder* class is defined as *LocalBinder*: it provides the *getService* method which returns the current instance of the service. This enables the *TripsActivity* to have direct access to the public methods of the service: for example, as shown below, the client calls the *setDataUpdateListener* method from the service.

```
public void onServiceConnected(ComponentName name, IBinder service
    ) {
```

```
    ForegroundOnlyLocationService.LocalBinder binder = (
   ForegroundOnlyLocationService.LocalBinder) service;
    foregroundOnlyLocationService = binder.getService();

    foregroundOnlyLocationService.setDataUpdateListener(
   TripsActivity.this);

    foregroundOnlyLocationServiceBound = true;
}
```

Listing 3.6: Activity calling setDataUpdateListener() method of the service

From the client's side, the *TripsActivity* component binds to the service by calling *bindService* in the *OnStart* method. As mentioned above, when the app is not visible to the user, the service starts running in the foreground and the client stops to be bound to the service. This implies binding in the *onStart* callback method, unbinding in the *onStop* callback method, and starting the service also by calling the *startService* method, as shown in the following lines.

```
 protected void onStart() {
    super.onStart();
    Intent serviceIntent = new Intent(getApplicationContext(),
   ForegroundOnlyLocationService.class);
    serviceIntent.putExtra(
   EXTRA_CANCEL_LOCATION_TRACKING_FROM_NOTIFICATION, false);
    startService(serviceIntent);
    bindService(serviceIntent, foregroundOnlyServiceConnection,
   Context.BIND_AUTO_CREATE);
}
```

Listing 3.7: Binding of the Service in OnStart() method

The *bindService* method monitors the connection with the service through *foregroundOnlyServiceConnection*, an implementation of *ServiceConnection*. When creating this connection between the client and the service, the *onServiceConnected* method is called by the Android System: it includes the *IBinder* argument above considered, the one the client can use to communicate with the bound service.

```
private final ServiceConnection foregroundOnlyServiceConnection =
   new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName name, IBinder
   service) {
    ForegroundOnlyLocationService.LocalBinder binder = (
   ForegroundOnlyLocationService.LocalBinder) service;
    foregroundOnlyLocationService = binder.getService();

    foregroundOnlyLocationService.setDataUpdateListener(
   TripsActivity.this);
```

```

    foregroundOnlyLocationServiceBound = true;
    }

    @Override
    public void onServiceDisconnected(ComponentName name) {
    foregroundOnlyLocationService = null;
    foregroundOnlyLocationServiceBound = false;
    }
};
```

Listing 3.8: Service Connection

3.4.2 Location Service as Foreground Service

As mentioned above, and shown in the lines below, the service starts running in the foreground, making users aware of their current position through the notification, when the client unbinds.

- Boolean *serviceRunningInForeground* set to *true*.

```
@Override
public boolean onUnbind(Intent intent) {
 Log.d(TAG, "Start foreground service");
 Notification notification = generateNotification(currentLocation
  );
 startForeground(NOTIFICATION_ID, notification);
 serviceRunningInForeground = true;

 return true;
}
```

Listing 3.9: Unbind() method

The foreground service is started by calling the *startService* method in the client's *onStart* callback method, and is stopped by itself in the service with the *stopSelf* method.

In this case, the service will be stopped when the user decides to cancel the Location tracking from notification:

- Boolean *cancelLocationTrackingFromNotification* (by default set to false) turns to *true*, as shown in the following lines.

```
    @Override
    public int onStartCommand(Intent intent, int flags, int
  startId) {
    Log.d(TAG, "onStartCommand()");

    if (intent.getBooleanExtra(
   EXTRA_CANCEL_LOCATION_TRACKING_FROM_NOTIFICATION, false) )
   {
      boolean cancelLocationTrackingFromNotification =
      intent.getBooleanExtra(
   EXTRA_CANCEL_LOCATION_TRACKING_FROM_NOTIFICATION, false);

      if (cancelLocationTrackingFromNotification) {
        unsubscribeToLocationUpdates();
        stopSelf();
      }
    }
    Location_at_Start();
    return START_NOT_STICKY;
}

```

Listing 3.10: onStartCommand() method

Every time the user goes back to the app and the client rebinds to the service, the notification disappears:

- Boolean *serviceRunningInForeground* set to false, as shown in the code below; (the *true* value returned by the *Unbind* callback method ensures the client rebinds to the service when goes back to the foreground).

```
     @Override
     public void onRebind(Intent intent) {
      Log.d(TAG, "onRebind()");

      stopForeground(true);
      serviceRunningInForeground = false;
      configurationChange = false;
      super.onRebind(intent);
    }

```

Listing 3.11: onRebind() callback method

Notification system

In order to post Foreground services *notification*, first of all, the required permission has been added to the *Manifest* file.

The notification content and the notification channel have been set using a *NotificationCompat.Build* object, through which the following parameters have been defined [28]:

- a small icon, set by *setSmallIcon* method;
- a title, set by *setContentTitle* method;
- the body text, set by *ContentText* method;
- the dimensions of the notification area, set by *setStyle* method, which enables the notification to be expandable;
- the notification priority set by *setPriority* method, defining the channel importance (before posting any notification, a notification channel is created by passing an instance of *NotificationChannel* to the *createNotificationChannel* method and, then, the channel ID is provided to the *NotificationCompat.Builder* constructor);
- the authorization for the notification to be automatically removed when the user taps it, set by *setAutoCancel(true)* method;
- an action button, set by *addAction* method (in this case a pending intent has been defined, so that, when the user taps the "stop receiving location updates" button, the service will stop itself).

3.4.3 Location Update Request

In order to advance location updates' requests, the *subscribeToLocationUpdate* method has been defined in the service.

After declaring the required permissions in the *Manifest* file,

ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION, the request and removal of location updates passes through the *FusedLocationProviderClient* initialization. When requesting location updates, the *FusedLocationProviderClient* takes in input a LocationRequest parameter, a *LocationCallback* parameter, and a *Looper* object which specifies the thread for the callback. The *LocationRequest* is a data object containing quality-of-services parameters for requests (intervals for updates, priorities, and accuracy). The *LocationCallback* is used to receive notification when the device location has changed or can no longer be determined. This is passed a *LocationResult* where the Location can be used to define the current location to send to the *TripsActivity* component. In order for the current location to be transmitted to the *TripsActivity* component, the *sendDataToActivity* method is considered. The *sendDataToActivity* method calls the *onDataUpdated* method in turn mentioned in the *DataUpdateListener* Interface. As shown in the code related to the *Service Connection*, the *TripsActivity* component is the *DataUpdateListener* element calling the *onDataUpdated* method: it means that, every time a new location update is available, the *TripsActivity* component will be notified by the service and will receive the updated current location. In order to remove the request for location updates, the *unsubscribeToLocationUpdate* method will be called.

Implementation of Location Update

By binding the app's component *TripsActivity* to a foreground service, is possible to request the update of the users' current location in the following cases (as required by the app's logic):

- When the user starts interacting with the *TripsActivity* component, in order to show the position on the Google Map before starting a trip.

  There is a direct call to the *subscribeToLocationUpdates* method in the *onStartCommand* method, in the service. In this case, once the location is notified to the *TripsActivity* component, it will be shown to the user in the form of a real address. It requires the implementation of a *geocoding* process, as in detail explained in the Section 3.4.4 and the integration of a map, as explained in Section 3.4.5.

- When the user presses the button for finishing the trip, in order to update the position on the map, and to process the number of trip's kilometers.

  The *Location_at_stop* method is called in the *TripsActivity* component and defined in the service: it simply takes the current value of the current position. It is assigned to the last location update, according to the *LocationRequest* parameters in the service.

- When a harsh event is detected.

  A SELECT query with a WHERE clause on the event timestamp is executed on the table collecting GPS data in order to retrieve the location corresponding to the "zero" harsh event.

- When the user navigates away from the app's activity, while the trip is active.

  A notification displaying the current position's coordinates will be shown, as already explained.

- When users go back to the app.

  The *onRebind* callback method will be called: it is ensured by the *true* return value of the *onUnbind* method.

3.4.4 Geocoding Process

Once obtained the position through localization, is possible to convert the latitude and longitude coordinates into a real address: the geocoder precisely plays this role. It is initialized in the *onCreate* method, by executing the *getFromLocation* method, which takes in input the coordinates of the position, saves into an empty list of addresses all the possible addresses associated with those coordinates. From this list, the country, the address, the postal code related to the detected position are defined by considering the first item (first result) of the addresses' list.

3.4.5 Map's Integration

In order to integrate the map, before working on the Android project, a Google project with specific settings was created into the Google Developers Console. More specifically, once created the Google project, the Google Maps Android API have been enabled and an API key (Android key) was created. When configuring the key in the Credentials section of the Google Console, a name for the key, the package used for the application, a SHA1 code obtained thanks to the keytool instrument, available into the JDK distributions were asked and defined. The result of the configuration was a long alphanumeric sequence to be used in the Android project.

At that point:

- the required permissions and the obtained key were introduced in the Manifest file;

- the map was inserted into the *Fragment_Map* layout.

About the programming code:

- The *Fragment_Map* component started to implement the *onMapReadyCallback* interface, in order to exploit the *onMapReady* callback method, in order to further use the *GoogleMap* object.

- In the *onCreate* method, the map was obtained through the *getMapAsync* method, which works in asyncrhronous mode: once the map is ready, a reference to the map is passed the *onMapReady* method.

- The *updateMap* method was inserted in order to update the position inside the map according to the location update requests.

As shown in the code below, in order to individuate the portion of the map to be framed, the *movecamera* method is implemented; it takes in input a *CameraUpdate* object which defines the new position where the map's camera should move. The position into the map is individuated by a marker, which is a placeholder for a specific location inside the Google Map. The marker is added through the *addMarker* method, specifying the position (the same where the camera moves into the map), and the title of the marker: in case the *updateMap* method is invoked when the user starts interacting with the activity, the title is set to “You are here!”; in case the *updateMap* method is invoked when the user stops a trip, the title is set to “You were here at the moment of the stop!”.

```java
public void updateMap(){
 LatLng ll = new LatLng(dataProvider.getPosition().getLatitude(),
   dataProvider.getPosition().getLongitude());

 CameraPosition pos = new CameraPosition.Builder().target(ll).zoom
   (14).build();
 map.moveCamera(CameraUpdateFactory.newCameraPosition(pos));
 Marker marker = map.addMarker(new MarkerOptions().position(ll));

 assert marker != null;
 marker.setTitle("You are here");
}
```

Listing 3.12: Update Map method

3.4.6 Device’s Orientation

Four methods can be taken into consideration in order to determine the orientation of the device, each one using different combinations of different kind of sensors:

- gravity sensor;
- accelerometer + magnetic sensor;
- gravity sensor + magnetic sensor;
- rotation sensor.

In order to overcome the limits related to the first three methods, in terms of low quality and reliability of data due to the noise related to the sensor’s hardware aspect, in this work, the technique of the *Rotation sensor* is adopted. The rotation sensor is a synthetic sensor obtained through Sensor Fusion algorithms, which allow to easily determine the device’s orientation, without worrying about the complexity and problems related to the hardware. By exploiting the Sensor Fusion between accelerometer, magnetometer and gyroscope, it produces a Rotation Vector which can be converted in a corresponding rotation matrix through the *SensorManager.gerRotationMatrixFromVector* method, as shown above, in the code related to the *doInBackground* method. Starting from the obtained rotation matrix, the orientation of the device will be determined through the *SensorManager.getOrientation* method, inside the *defineOrientation* method.

The *SensorManager.getOrientation* method takes in input two parameters: a rotation matrix and a float array consisting of three values measured in radians, representing the azimuth, the pitch and roll (see code below).

The *azimuth* refers to rotation around a vertical axis: imagine turning left and right like a compass needle.

The *pitch* refers to rotation around a longitudinal axis: imagine tipping forward and backward like a seesaw.

The *roll* refers to rotation around a lateral axis: imagine something rolling around the ground like a wheel.

From these three values, pitch and roll are then considered to determine the *face up/down.* Even if the pitch should be equal to zero in case the device is positioned perpendicularly with respect to the ground, eventual noises must be taken into consideration: a threshold value equal to 10 is considered for the pitch in the algorithm. Same reasoning should apply for the roll, but, as is not crucial to know the sense of the device's rotation, the threshold value is compared to the absolute value:

- *face Up*, when the roll's absolute value is equal or lower to 10 grades;

- *face Down,* when the roll's absolute value is equal or higher than 170.

Once the device's orientation is determined, the *onFaceUp / onFaceDown* methods are defined to notify it, as shown in the following lines.

```java
private void defineOrientation(float[] rotation_matrix) {
 float[] orientation_Values = new float[3];
 SensorManager.getOrientation(rotation_matrix, orientation_Values)
   ;

 double azimuth = Math.toDegrees(orientation_Values[0]);
 double pitch = Math.toDegrees(orientation_Values[1]);
 double roll = Math.toDegrees(orientation_Values[2]);

  if (pitch <= 10){
    if (Math.abs(roll) >= 170){
        onFaceDown();
    } else if (Math.abs(roll) <= 10) {
        onFaceUp();
    }
  }
}
```

Listing 3.13: Define Orientation method

```java
private void onFaceUp() {
        Log.i(TAG, "The screen of the device is facing Up!" );}
private void onFaceDown() {
        Log.i(TAG, "The screen of the device is facing Down!");}
```

Listing 3.14: Face Up/ Face Down

3.5 Blockchain

As already mentioned, one of the aims of this project is offering the app's users the opportunity to interact with Blockchain. The interaction with the Blockchain platform is directly requested by the application's users when performing tasks on which the Gamification's logic is built, by calling specific methods of the Smart Contract. In order for users to write or read data from Blockchain, a connection between the Mobile App and the Blockchain platform is defined, as explained in the next section.

3.5.1 Interaction with Blockchain

The interaction between the Mobile App and Blockchain is based on the use of *Web3J*, *Infura* and *Sepolia* Test-net.

Web3J is a java library that enables to create decentralized java applications without having to write integration code for the Blockchain platform. In this project, the *web3J* library is adopted to create *Ethereum wallets*, to get wallet addresses, retrieve balance and to load the deployed Smart Contract, in order to call its methods.

Before using the *web3j* library, a connection with the *Ethereum* network is opened by creating an *Infura API key* [20]. *Infura* provides the opportunity to connect to an *Ethereum* network and carry out transactions without having a local node. For creating an *Infura API*, after going to the website, an account and a project were created. Once a new project was created, a copy of the endpoint *URL* was saved. This endpoint is used to send API requests from the decentralized application and it represents the main connection point with the Ethereum Blockchain: when creating an instance of the web3j class, this *URL* is needed.

Implementation of Web3j in the Android Project

To implement *Web3J* in the project, at first the third party *web3j* library dependency was added in the *build.gradle* file, and the required Internet permissions were added to the *Manifest* file.

To connect the application with the *Ethereum* network, the *build* method is invoked into the *init_web3* method, into the *TripsActivity* component. The build method takes in input the *httpservice* and connects to the Ethereum network using the *Infura API key*. In this case, the application is connected to the *Sepolia* testnet.

After creating an instance of the *web3j* class, the *verify_connection* method and the *setupBouncyCastle* method are called in the *OnCreate* callback method. The *verify_connection* method simply tests the connection to the *Sepolia* network, by notifying whether or not the connection is successful. About the *setupBouncyCastle* method, the *Bouncycastle* cryptopackage implements cryptographic algorithms needed for the Blockchain security. The Security class centralizes security properties and also manages providers. For *web3j* the ECDSA algorithm is implemented by the security provider. To get the provider, the *bouncycastleprovider* is passed to the *getprovider* method.

Creating an Ethereum Wallet

An *Ethereum wallet* requires a file path and a password. Through the *WalletUtils* class, the methods used to work with Wallet files are:

- *Loadcredentials* method: it takes in input the password inserted into an EdiText widget by the user, and the file path. It is used to get credentials of the given wallet's path and password.

- *GenerateLightNewWalletFile* method: by taking in the password inserted into the EditText widget by the user and the file path, it generates the Ethereum wallet and the json file. The file path is obtained by considering the name of the wallet directory inserted by the user into an EditText widget.

It's worth to specify that, when a new wallet is created, all the data (directory's name, password, wallet address and json file) are saved into the application's database. For this reason, before a new wallet is generated, starting from the data related to the wallet's password and the name of the wallet's directory, inserted by the user into the two EdiText widgets, two SELECT queries with a WHERE clause are executed on the table Users. If the cursors returned by the queries are not empty, it means that the data provided have been already inserted into the database, so that the wallet corresponding to those values is already existing. If the cursors are empty, the new Ethereum wallet is generated. Then, through the *getAddress* method the wallet address will be retrieved and displayed in the UI.

Retrieving Balance

To get the wallet's balance, the address and the last *DefaultBlockParameterName* are passed to *ethGetBalance* method. Then, to get the balance, the *getBalance* method of the *EthGetBalance* class is used.

3.5.2 Smart Contract

As above mentioned, the Smart Contract was developed to define the Gamification's logic. The aim of this project, based on the contributions given by the Mobile App and this Smart Contract, is to design a raw system model able to safety evaluate the driver's behaviors and to stimulate drivers in being monitored from one side, and in approaching a safe driving style from the other side. Knowing that this project is meant to be developed together with Insurance Companies (ICs), the system model defined in this work would like to overcome the barriers still faced by ICs with respect to the model of the proposed Personalized Car Insurances (PCI), one of which is the lack of transparency, i.e., the rules of PCI are not clear and how drivers' auto premiums are calculated based on their behavior is unknown. If transparency of data and processes is a goal from one side, the privacy preservation of the drivers is a goal from the other side. How to avoid data cheating? Which data can be shared and stored on the Blockchain to achieve both transparency and privacypreservation? How to stimulate drivers in approaching a safe driving style?

The answer to these questions is the base on which this system model has been designed and the design goals have been defined.

3.5.3 System Model

The system model consists of:

- *Drivers*: they are the application's users and the *Players* into the Smart Contract.

- *Gamification mechanism*: the Gamification's logic is defined into the Smart Contract, whereas data related to the trips' scores (to be processed into the Smart Contract) are sent by the Mobile App.

As in this work data used and processed in the Smart Contract derive from the Mobile App's processes, is interesting to highlight how data are treated in the two different contests in order to fulfill the design goals: transparency and driver privacy-preservation. It is illustrated by answering to the mentioned relevant questions.

How to avoid data cheating?

By considering the contribution given by the Mobile App, if considering the implemented programming code, there is no opportunity for the app's users to cheat driving-data or to select the driving-data to be used in the computation of the trip's score. When the user presses "Finish Trip" button, all the data collected in background start to be analyzed, the number of harsh events is detected, and the *getYourTripScore* method is automatically called. This process of computing the trip's score is described in Section 3.3.4. Once the trip's score is calculated, the *load_Trip* method of the Smart Contract is called, so that the trip's score will be written and stored into the Smart Contract. With the aim of avoiding cases of misleading trip's data, in the score's computation process the number of kilometers actually traveled on the trip is calculated, and the impact of the harsh events is distributed over the total driving distance. It implies that, for a rational driver, there is no reason to think "the less I travel, the lower the probability to commit harsh events, the higher the score": there is a minimum distance for the trip to be considered as valid, and, in case of same harsh events' number, the shorter the trip, the lower the score. Then, by making the transfer of "sensitive" data (the ones relevant to the Gamification context) automatically invoked by the Mobile App's programming code, the probability of data cheating is significantly reduced.

Which data can be shared and stored on the Blockchain to achieve both transparency and privacy-preservation?

From the perspective of the drivers, the concept of transparency can be intended as related to the algorithm implemented in the score's computation process of each trip, to the algorithm implemented for defining the players' ranking inside the Gamification context, and to the data processed in such algorithms. Starting from the last point, as considered above, by trusting the implemented programming code, the data processed inside the score's computation process, at the level of the Mobile App, derive from the ones collected in background by the sensors, then filtered in order to remove the noise and gravitational forces, then saved into the application's database, and finally processed in order to first count the harsh events and then define the trip's score. The values detected by the sensors and destined to be processed are saved into the database. The harsh events defined by processing the sensor's data are saved into the database as object of the Classes *Hard_Acceleration_Event*, *Harsh_Braking_Event, Harsh_Cornering_Event.* An instance of these Classes is an object characterized by the timestamp of the harsh event, the localization of the harsh event, and the value exceeding the threshold that defined the event. The reason behind these three Classes is to make the application's users and the Smart Contract's players trusting the evaluation process of their driving behavior: this idea to associate the position where the abrupt maneuver was committed to the harsh event detected is a way to ensure transparency, because the sensitive data are verifiable and visible to the users. About the transparency of data stored on the Blockchain platform, each player is allowed to look at the scores achieved by the other players. Moving to the transparency of the implemented algorithms:

- At the Mobile App level, in the computation process of the score, the algorithm is literally explaining in one of the voices of the Drawer menu. The aim is to make users aware of the way their driving-data will be processed.

- At the level of the Smart Contract, the Gamification logic is defined by the methods included into the Smart Contract which is deployed on the Blockchain, for this reason accessible and visible by everyone.

As regard the privacy-preservation, at the level of Mobile App, users are allowed to access only those data related to their own trips while trips' relevant data of other users are not accessible. At the level of Blockchain, the privacy is preserved by making public only the scores achieved, without publishing the driving features leading to those results. Furthermore, whereas in the Mobile App's database every user is registered with personal data (first name, surname), in the Smart Contract each player is only identified by a nick name.

In line with this revision of public/private data when passing from the Mobile App to the Smart Contract in order to achieve the privacy-preservation goal, the Table shown by the Table 3.1 has been drawn, where:

- for the Mobile app, the "*Public*" word means that every user can access those data, and the "*Saved in DB*" expression means that only the user to whom data are related can access those data;

- for the Smart Contract, the "*Public*" word means that those data are uploaded and registered on the blockchain platform, whereas the "*Not Uploaded ('-')*" expression is associated to sensitive data not uploaded on the blockchain for privacy preservation.

As shown by the table 3.1, the comparison is focused on those variables considered to be relevant for the drivers (personal data) and for the algorithms defining the Mobile App's logic and the Gamification's logic in the Smart Contract.

How to stimulate drivers in approaching a safe driving style?

As already mentioned, this project is meant to be developed together with Insurance Companies. By focusing on the way drivers can be stimulated by ICs to approaching a safe driving style, it can be reasonable to think about the implementation of further discounts on the auto premiums that will be adjusted according to the evaluation results. Moving the focus to the thesis work, a Gamification mechanism has been designed to lead drivers toward a safer driving style: at the end of the game's period, the drivers getting the first three places of the final ranking will be rewarded with prizes based on the funds collected for that game. It implies the transfer of funds from the drivers to the Smart Contract when signing up for the game, and transfers of funds from the Smart Contract to the first three players at the moment of game closing.

In the next section the Gamification's logic will be described in detail.

Variables	Mobile App	Smart Contract
Driver's Name	Public	-
Driver's Surname	Public	-
Driver's Nickname	Public	Public
Driver's Wallet Address	Public	Public
Driver's Wallet Password	Saved in DB	-
Trip's Score	Public	Public
Trip's Destination	Public	-
Trip's Timestamp	Public	-
Trip's number of Harsh Acceleration events	Saved in DB	-
Trip's number of Harsh Braking events	Saved in DB	-
Trip's number of Harsh Cornering events	Saved in DB	-
Localization of Harsh Acceleration events	Saved in DB	-
Localization of Harsh Braking events	Saved in DB	-
Localization of Harsh Cornering events	Saved in DB	-
Timestamp of Harsh Acceleration events	Saved in DB	-
Timestamp of Harsh Braking events	Saved in DB	-
Timestamp of Harsh Cornering events	Saved in DB	-
Weight of driving features	Public	-

Table 3.1: Comparison about variables' visibility between the Mobile App and the Smart Contract

3.5.4 Gamification Context

In this section, the logic of the Gamification context will be described.

There are two possibilities for the application's users to sign up for a game, then to become *Players* of the Smart Contract:

1. *Start a new game*: i.e., "even in case of already existing games, you want to start a new game with your friends". In this case the *Init_Game* method of the Smart Contract is called by the Mobile App (through the wrapper file): this method takes in the nickname of the player who is setting the game, who is also required to give a name to the game, required to recognize it among

the others available games.

2. *Join an existing game*: in this case the *addPlayer* method of the Smart Contract is called and takes in, as before, the player's nickname and the game's name the player wants to join.

It's worth to specify that initializing a game does not mean starting a game: every initialized game will be set to started equal to *true* only when the number of players of the game has reached a value at least equal to *N_MIN_PLAYERS*. Starting from this consideration, until the number of players is lower than *N_MAX_PLAYERS* there will be space available in the game. Once it reaches the maximum number of players, the game's *spaceAvailable_completed* will be set to *true*. Furthermore, when a user plans to join a game, by passing the game's name to the method, it will be verified that in the Smart Contract there is a game corresponding to that name. If successful, the user becomes a player of that game.

Then, all the funds sent to the game by the game's players, are collected and saved into the game's *total_amount*. This amount is used to define the game's first prize, second prize and third price, as percentages of the whole amount. It's interesting to specify that funds are sent by users in *ether* value and are saved into the Smart Contract in *wei* value.

Once the game is set to started equal to *true* (the minimum number of players is reached), data about trips can be uploaded on the Blockchain. By preserving driver's privacy, only the trip's score and the game's name will be passed to the *load_Trip* method when invoked by the *stop_Trip* method from the Mobile Application. All the other data related to the journey, i.e., trip's destination, trip's timestamp, are saved into the application's database but not written into the Blockchain.

With the aim to define a current ranking, that will become the definitive final ranking at game closing, the *Total_Score* method is defined into the Smart Contract: by scrolling the scores of the trips uploaded by each of the game's player, the total score, given by the sum of each of the uploaded trips' scores, will be set for every player of that game. At the moment of game closing, the players ranked in the first three places of the final ranking will be rewarded with the game's first prize, second prize, and third prize: to transfer funds from the Smart Contract to the players' addresses, the *sendPrize* method is defined.

About the game's time window, a game can remain open for a maximum of one year: the expiration date is defined when the minimum number of players joining the game reaches the required value. That moment sets the *startAt* date and the *expiresAt* date for that game. In case, while the game is open, one of the players wants to retire, the *Retire_From_The_Game method* is called; it takes in a string defining the motivation behind the retirement and the game's name from which the player wants to retire. The retirement implies the transfer of the subscription fee back to the user, and the review of the game's total available amount with the game's prizes. In case, at the time of game closing, some funds corresponding to that game are left, they will be transferred to the address corresponding to the Smart Contract's deployer.

In the table 3.2, a summary of the functions defined in the Smart Contract.

Functions	**Description**
Init_Game	Initialization of a new game.
addPlayer	A new application's user is joining an existing game.
load_Trip	The trip's score is uploaded into the blockchain.
First_Player	The player positioned at the first place of the final ranking is determined.
SecondPlace	The player positioned at the second place of the final ranking is determined.
ThirdPlace	The player positioned at the third place of the final ranking is determined.
sendPrize	The prizes are transferred in wei value from the Smart Contract to the Players ranked in the first three places at the moment of game closing.
Retire_From_The_Game	One of the game's players want to retire from the game. It implies transfer of refund, resetting of the game's total amount and prizes.
Close_Game	The expiration date has come. If some funds are still available for the game, they are transferred to the Smart Contract's deployer.

Table 3.2: Smart Contract's Functions

Chapter 4

User Experience

This chapter aims to describe the project's products, Mobile App and Smart Contract, taking the perspective of a new user who explores and then interacts with the product. Starting from the View elements by which the user directly interacts, he will be able to know and easily understand the main features that the app makes available to its users. Starting from the User Interface (*UI*) level, the rules through which the application creates, modifies and stores data will be identified. This chapter will therefore describe the two main levels of an app's architecture starting from the user experience: from *UI* level to *Data layer*, which includes the business logic of the application and the internal data structure, intended as Classes and Local Databases [29] [30].

Taking the user's perspective, at first, the *IDEF0 Analysis* is proposed with the aim of bringing users to become familiar with the logical structure of the application.

4.1 IDEF0 diagrams

The IDEF0 analysis is carried out by presenting two models (as shown in Figure 4.1 and Figure 4.2):

1. the A-0 diagram, also called Top-Level diagram through which users can easily understand how the application works by looking at the main functionalities they can get access;

2. the A0 diagram, where all the elements shown in the A-0 diagram are shown again but the app's whole logic is here split into seven main functions.

Focusing on the A0 Diagram, as shown in the Figure(4.2), the very first function required is obviously dealing with the sign up. This allows users to subscribe to

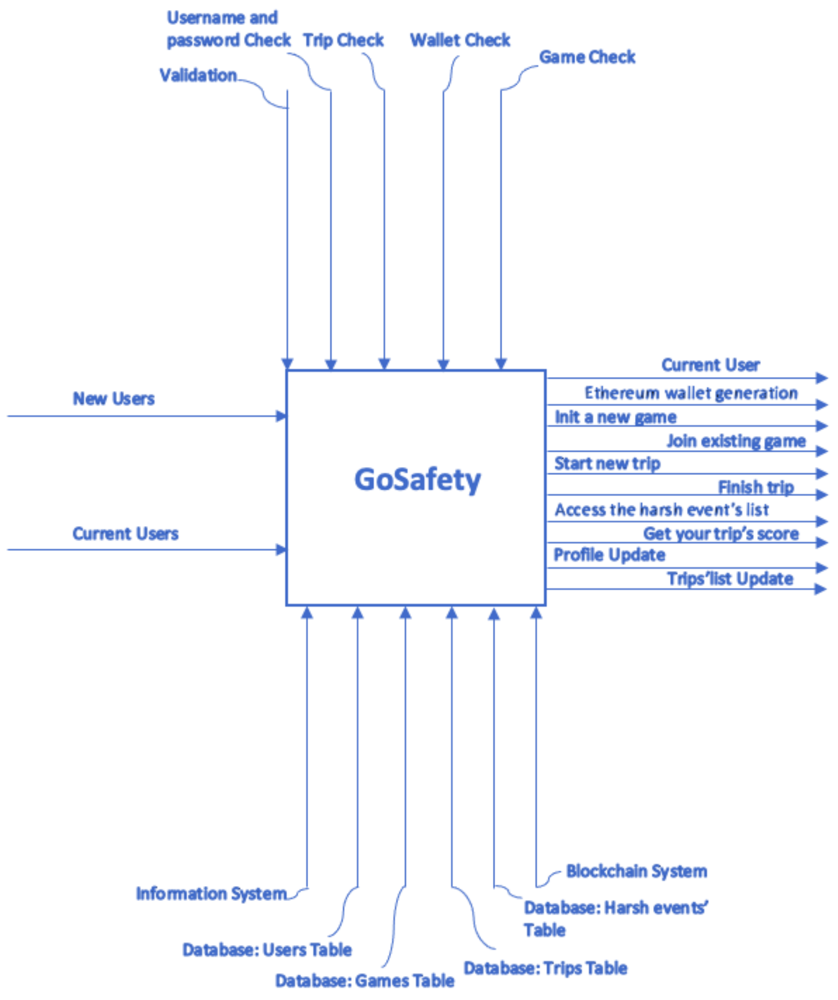


Figure 4.1: IDEF0: GoSafety

the application, and once logged in, to become current and "activE" users. Once reached the *TripsActivity* Page, they can decide:

- to start a new trip;
- to finish a current trip;
- to create an Ethereum Wallet;
- to initialize a new game;
- to join an existing game.

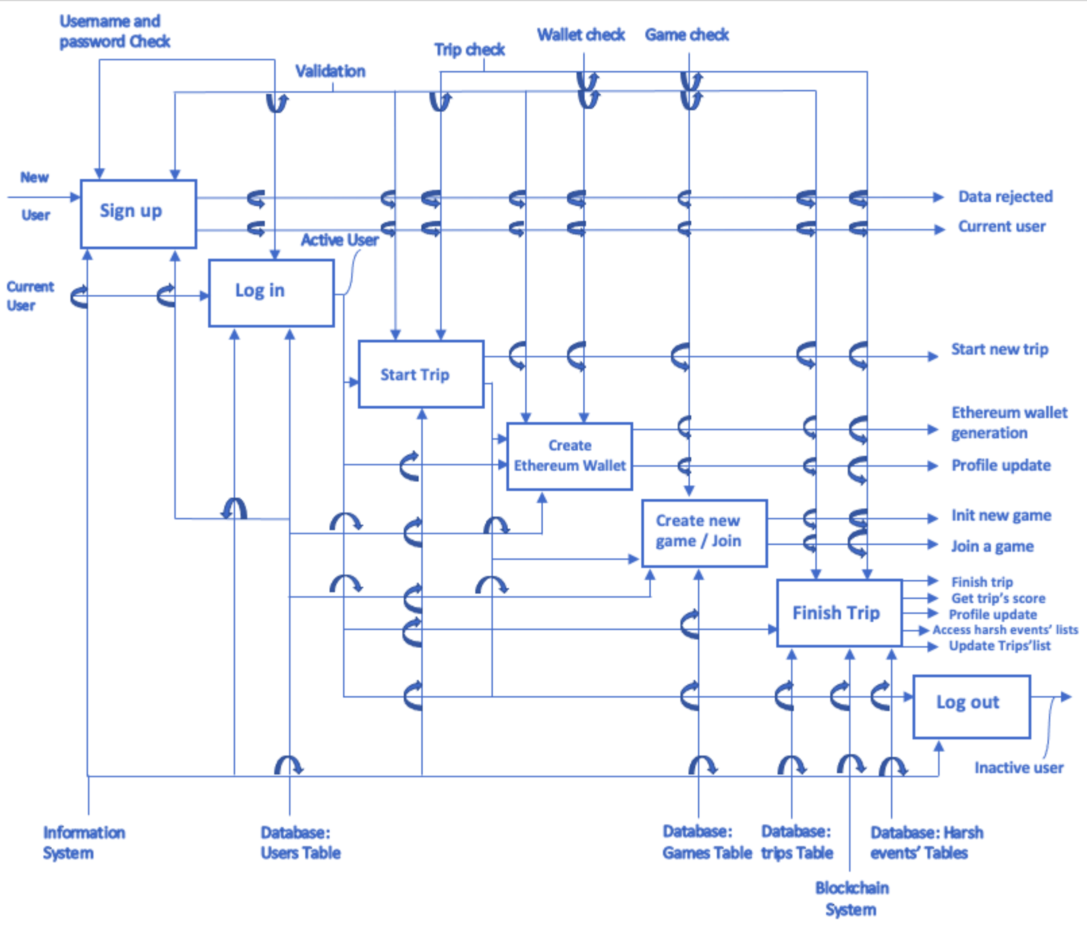


Figure 4.2: IDEF A-0: GoSafety functionalities

Furthermore, moving to the drawer menu and scrolling its voices, they can check personal information saved into the database at the moment of the registration phase, they can update their own personal profile page, they can look at all the users involved in any games, at all the available games, and they can access the harsh events committed during trips already done. The drawer menu, that will be described in detail in the section 4.2.3, also includes the log out function, which turns active users into "inactive" ones.

4.2 UI Level

Before going in the details of the activities composing the User Interface (*UI*), a brief introduction about the UI's role is proposed [29].

The term UI refers to elements such as Activities and Fragments displaying application data: as mentioned, its role is to display data on the screen and also to serve as the main point of user interaction. However, the application data obtained by the database is usually in a different format than the information to be displayed. Because the data layer's role is to hold, manage, and provide access to the application data, the UI layer must perform and repeat the following steps, as long as necessary:

1. consume app data and turns it into data that the UI can easily render;
2. consume UI-renderable data and turns it into UI elements to present to the user;
3. consume user input events from those assembled UI elements and reflect their effects in the UI data as needed.

Furthermore, whenever data changes, either due to user interaction (such as a button press) or external input (such as a network response), the user interface should update to reflect those changes in the UI status. In other words: if the UI is what the user sees, the state of the UI is what the app says they should see. Like two sides of the same coin, the user interface is the visual representation of the state of the interface. Any changes to the UI state are immediately reflected in the UI.

About the Activities and Fragments, they provide the window in which the app draws its UI. This window typically fills the screen, but may be smaller than the screen and float on top of other windows; generally, one Activity implements one screen in an app, and an app contains multiple screens, meaning that it comprises multiple Activities. The way Activities are launched and put together is a fundamental part of the platform's application model. Unlike programming paradigms in which applications are launched with a *main()* method, the Android system initiates code in an *Activity* instance by invoking specific *callback methods* corresponding to specific stages of its life-cycle:

- *OnCreate(Bundle)* is where the Activity is initialized. Most importantly, here the *setContentView(int)* method is usually called with a layout resource defining the Activity UI, and using *findViewById(int)* to retrieve the widgets needed to be used in the programming code.
- *OnPause()* is a callback method dealing with the user pausing active interaction with the Activity. Any changes made by the user should at this point be committed (usually to the *ContentProvider* holding the data). In this state the Activity is still visible to the user on the screen.

To be of use with *Context.startActivity()* activity classes must have a corresponding <*activity*>declaration in their package's *Androidmanifest.xml*.

Moving the focus back to the User Interface of the project's Mobile App, users have the possibility to access the above described app's functionalities by interacting with the following activities, which will be presented one-by-one in the next sections:

- *landing* Page (4.2.2);
- *registration* activity, included in the section 4.2.2;
- *login* activity, included in the section 4.2.2;
- *tripsActivity* component, which includes two *Fragments* (4.2.3);
- *drawer menu* (4.2.3).

Each of the activities is described starting from the Activity Diagram (UML Diagram) and showing the layout of the page, which includes all the View elements through which users can interact with the application and get access to all the functionalities.

4.2.1 UML: Activity Diagrams

The UML, that is the acronymous for *Unified Modeling Language*, is a graphical language used to write in a standardized way a software system's blueprint (in the field of OOAD, object oriented analysis and design). It is a useful tool to visualize, specify, construct and document the artifacts of an object-oriented system, showing the structures and the relationships within complex systems. In this analysis, the following UML diagrams are proposed:

- *Activity Diagram* for each of the Activities building up the application: it describes the sequence of actions in a process, showing in details how an Activity is subdivided in simpler actions. However, it does not only show actions that are subsequencial, but also mutual exclusive actions, or concurrent actions that must be performed together.
- *ERDs or ER Diagrams*, to explain the application's data structure of the Local Database, also introducing some references to the project's Classes of objects (4.3.1).
- *Sequence diagram*, developed for explaining how the process of detecting the users' driving-behavior is implemented by the code and illustrated to the users (4.4).

4.2.2 Landing Page

As shown in the Figure 4.4, The Landing Page Activity gives the welcome to users. By pressing the button element, the users will be directed to the Login Activity, which layout is presented in the Figure 4.5 below.

Taking into consideration the Activity Diagram 4.3, the user can easily understand that, once reached the Login Page, it is possible:

1. to directly insert data for *Login*, if the user has already created an account;
2. to press the button element to be directed to the *Sign Up* page, which layout is presented in the Figure 4.6, if the user is going to be a new active user.

In both cases, once the data required have been inserted, they will be checked in order to be verified (Login activity), or in order to be validated and verified (Sign Up activity). The validation rules for the registration phase relate to the format of data inserted, whereas the username and password checks ensure that data inserted have not been already chosen by a different user. In the Login phase, the username and password checks simply verify that data correspond to a real registered user. These checks are implemented by means of queries to the application Database, that will be deeply described in a specific section (4.3). In case of successful data validation and checks, from the Sign Up activity the user will be redirected to the Login activity. In case of successful data checks in the Login activity, the user becomes an *active user* and will be directed to the *TripsActivity* activity, where will initialize the *currentUser* instance of the User Class. Conversely, in case of unsuccessful validation and checks of data, the user is notified to reinsert data.

In Figure 4.4, the layout of the Landing Page is shown.

In Figure 4.5 the layout of the Login Page is shown, while in Figure 4.6 the layout of the Sign Up Page is given.

4.2.3 TripsActivity Page

Before moving the focus to the logic behind this activity, the activity's User Interface is described. As shown in Figure 4.8, this activity's UI is divided into two *fragments* [31].

- *What is a Fragment?*

  A Fragment represents a reusable portion of the app's UI, and introduces the concepts of modularity and reusability as the Activity's UI results to be divided into discrete chunks. It defines and manages its own layout, has its own life-cycle, and can handle its own input events. However, Fragments can't live on their own: their life-cycle unfolds hand-in-hand with that of the containing Activity. Then, they must be hosted by an Activity or another

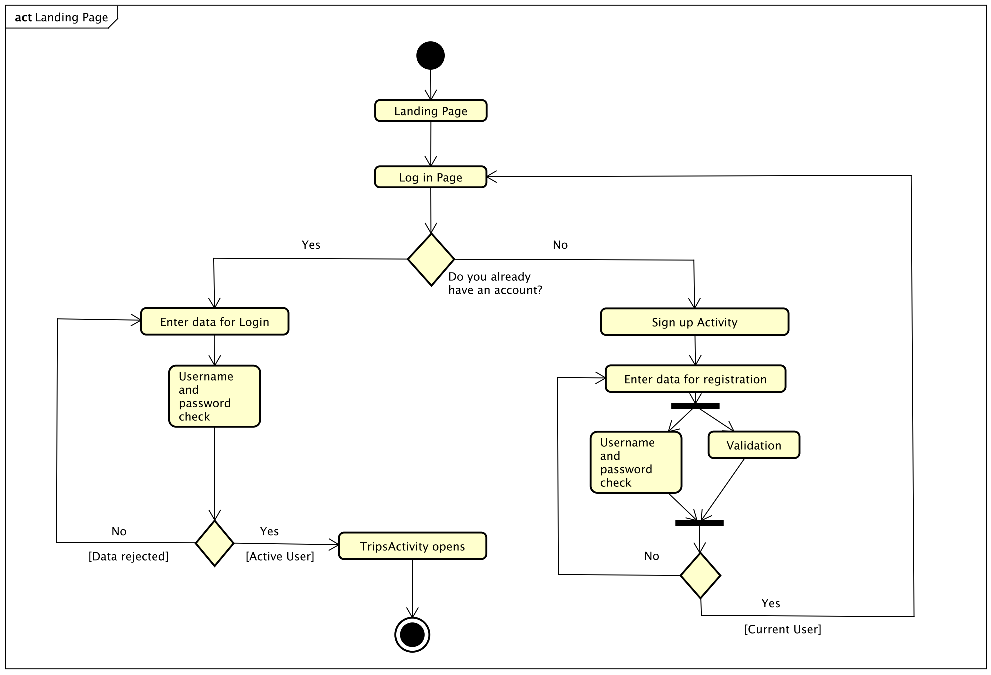


Figure 4.3: Activity Diagram: Landing Page

Fragment and the Fragment's view hierarchy becomes part of, or attaches to, the host's view hierarchy.

In the *TripsActivity* UI, there are two fragments:

1. a fragment with title "Ready" showing a map and managinge the whole process of detection of the drivers' behavior;
2. a fragment with title "Your Trips" showing the list of trips of the current user.

Within the project, the first fragment is associated the *Fragment_Map Class*, the second fragment is associated the *Fragment_TripsList* Class. Each Class defines the methods required to manage the specific layout portion of the Activity related to the Fragment.

Among the various configured methods, the following methods are relevant in terms of Fragment's life-cycle:

- Method *onAttach:* it is invoked when its reference *Context* is passed to the Fragment for the first time. Typically this is the moment when the Fragment knows its container Activity.

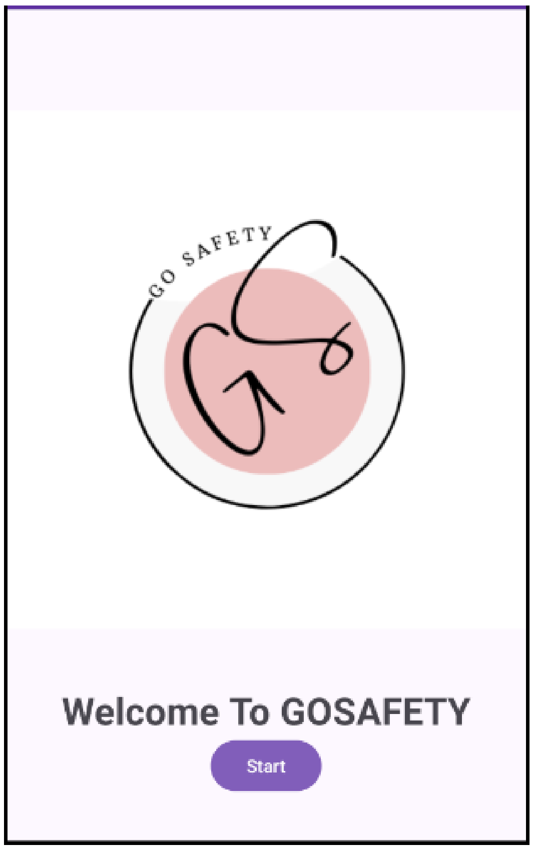


Figure 4.4: Landing Page Layout

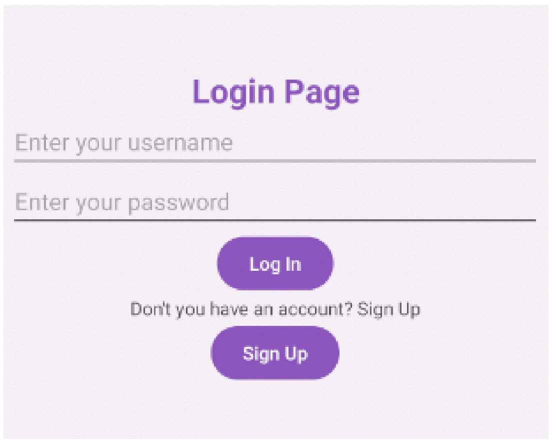


Figure 4.5: Login Page layout

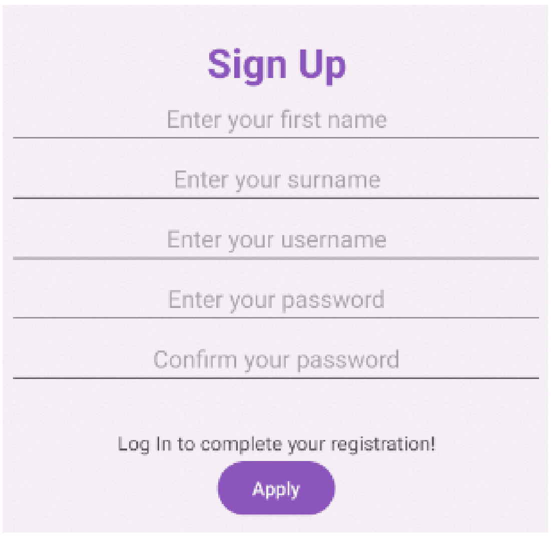


Figure 4.6: Sign Up layout

- Method *onCreateView*: it is used to create the layout of the Fragment. This method returns a View that contains the appearance shown by the Fragment.
- Method *onActivityCreated:* it notifies the completion of the creation of the container Activity. It is a relevant method: the Activity and the Fragment are created at the same time, therefore in the methods called before *onActivityCreated* a complete Activity is not always available.

- *ViewPager component*

  To create a multi-page layout, that users can scroll with a horizontal movement of the finger on the display (*swipe*), a component called *Viewpager* is used [32]. The pages composing it are the Fragments, which are managed by an adapter. To start using a *Viewpager*, it must be inserted into the layout with a specific XML node: *<android.viewpager.widget.ViewPager>*.

- *Fragment Adapter*

  There are two types of Fragment adapters:

  1. *FragmentPagerAdapter*: it manages all the Fragments representing the pages of a *ViewPager* and keeps all of them in memory: for this reason, this kind of adapter is preferably used in cases where there are not many Fragments to manage [33].

2. *FragmentStatePagerAdapter*: unlike the previous one, it destroys all the Fragments relating to the pages no longer visible to the user; for this reason it is adopted in cases where the number of Fragments to be managed is high.

Since there are two Fragments to be managed within the developed application, a *FragmentPagerAdapter* has been adopted, which is associated the *PagerAdapter* Class.

- *TabLayout*

  It is a component often used in conjunction with a *ViewPager* [34] [32]. This component allows to organize Fragments in a sort of filing "cabinet": each Fragment is marked with a label, so that it is possible to scroll the pages not only through the horizontal swipe. In this application, the *TabLayout*'s usefulness is not only linked to the "physical" structuring of the User Interface, but it also guarantees a hierarchical organization of the contents. Every time users finish their trip by pressing the "Finish your Trip" button element in the first Fragment, the trip will appear in the trips' list in the second Fragment.

  To insert the *TabLayout* component into the layout and to synchronize it with the *ViewPager*, the following steps have been performed:

  - physical insertion of the *TabLayout* in the layout file (*trips.xml*);
  - association of the *TabLayout* with the *ViewPager* in the Activity's java file (in *OnCreate*);
  - connection of the *PagerAdapter* to the *Viewpager* through the *ovverride* of *getPageTitle* method, which provides the titles of the individual pages.

Focusing now on the Activity Diagram, once reached the *TripsActivity* Page, the user can choose among scrolling the drawer menu, staying in the *Fragment_Map Fragment*, or swiping to the *Fragment_TripsList* Fragment. As regard the Drawer menu, the section 4.2.3 is provided. As regard the Fragment showing the trips' list, as above said, every time a user presses the "Finish Your Trip" button element in the Map_Fragment Fragment, the list of the user's trips will be updated and it will appear in the trips' list. As regard the Map_Fragment, which displays a Google Map showing the current position of the user through a Marker, if the user wants to start a new trip, the first requirement is the insertion of the trip's destination. At that time, by pressing the "Start your Trip" button element, a query on the application's database is executed in order to verify if the user already has his own wallet. If the user does not have any Ethereum wallet, a first dialog window will appear asking the user to insert the data required for the wallet's generation. However, before getting access to the wallet's generation process, a validation of the connection with the Blockchain platform is requests: if the connection results to be successful the user can get the Ethereum wallet, otherwise the process will stop. Once got the wallet, it is possible to press again the button to start a trip.

If the user already has his own Ethereum wallet, after a first step checking information about the Ethereum Wallet, a second query on the application's database is executed in order to verify if the user results to be already registered for a game. In case he/she is not yet registered for any game, a dialog window will open asking the user to choice between joining an existing game or initializing a new game. In case the choice is to open a new game, a Game check will be performed about the name the user defines for the game, and about the participation of the user in any different game. In case the choice is to join an existing game, the Game check will be executed about the presence of the user in any different game, and about the space availability of the selected game. Anyway, the success of these "transactions" also depend on the funds availability on the user's wallet address: when a new game opens, and when a user joins an existing game, the related data will be written on the Blockchain. If users do not have enough funds for making those transactions successful, the process will stop.

Once they get through those steps, they can start a new trip.

When the "Stop your Trip" is pressed by users, a final check over the trip is executed. There is a minimum distance to be travelled for making the trip valid, otherwise it will not be saved on the Database. If valid, the user's trips' list will be updated, and the trip's data will start to be analyzed to identify eventual harsh events and to determine the trip's score. At that time, after a successful check over the connection with the Blockchain platform and a successful check about the funds availability for *loading the trip* over the Blockchain Platform, the trip's relevant data for the Gamification mechanism will be written on the Smart Contract.

In Figure 4.8 the layout of this Activity is given, while in Figure 4.9 is represented the Drawer menu layout.

Drawer menu

The drawer menu is one of the two portions building up the *Navigation Drawer*.

The Navigation Drawer is a scrolling panel intended to contain commands for navigating the application. It is a menu that appears scrolling, typically from the left side of the display, consisting of two parts: a header menu (at the top) which acts as the *NavigationDrawer* header and a drawer menu which acts as an options menu. The options menu integrates those contents that would not find space in another component, the *ActionBar*, thus allowing the limitations of space and flexibility of the *ActionBar* to be compensated with its dynamism.

To insert this component into the project, the following steps were performed:

- Creation of a layout file (activity_main_drawer.xml) with insertion of the

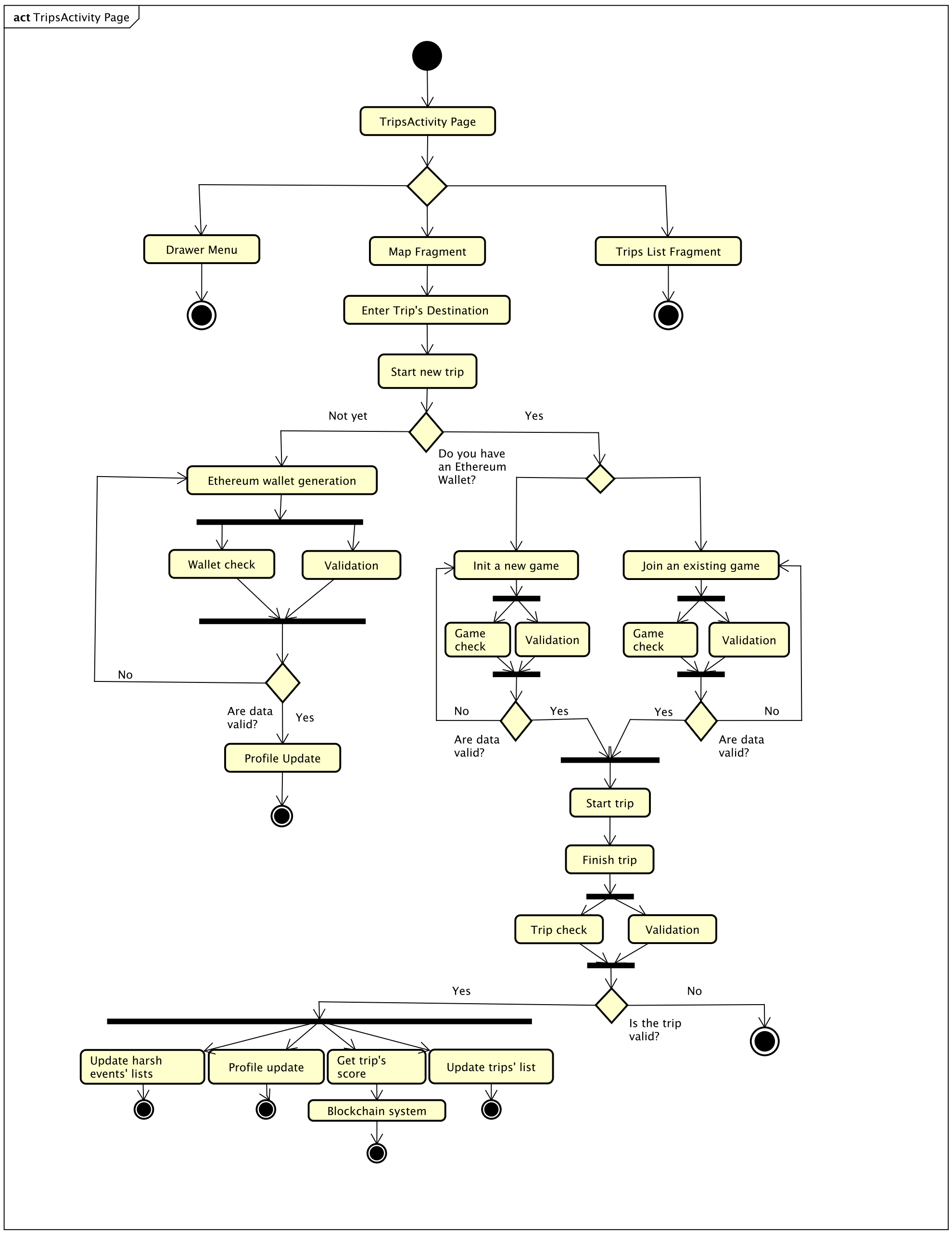


Figure 4.7: Activity Diagram: TripsActivity

main root node
<androidx.drawerlayout.widget.DrawerLayout>.
The layout drawer will become the main layout file of the application: the

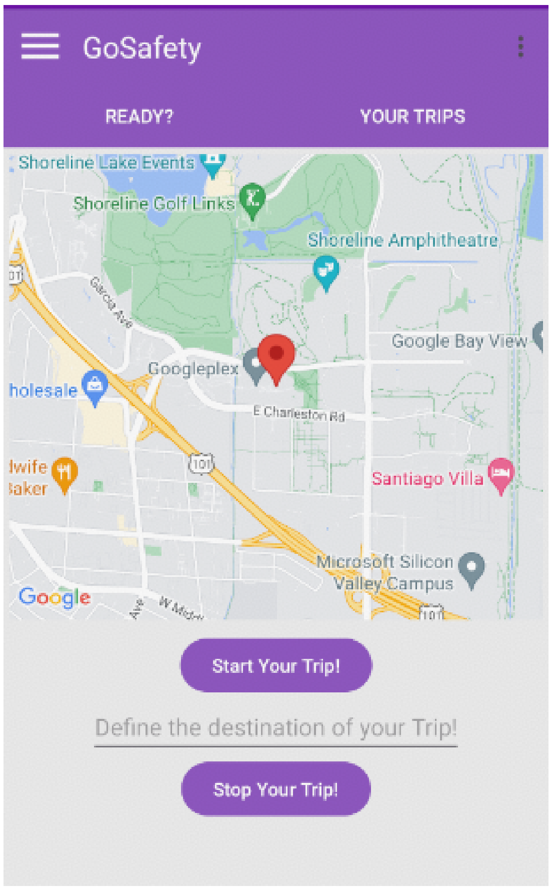


Figure 4.8: TripsActivity layout, with Fragments

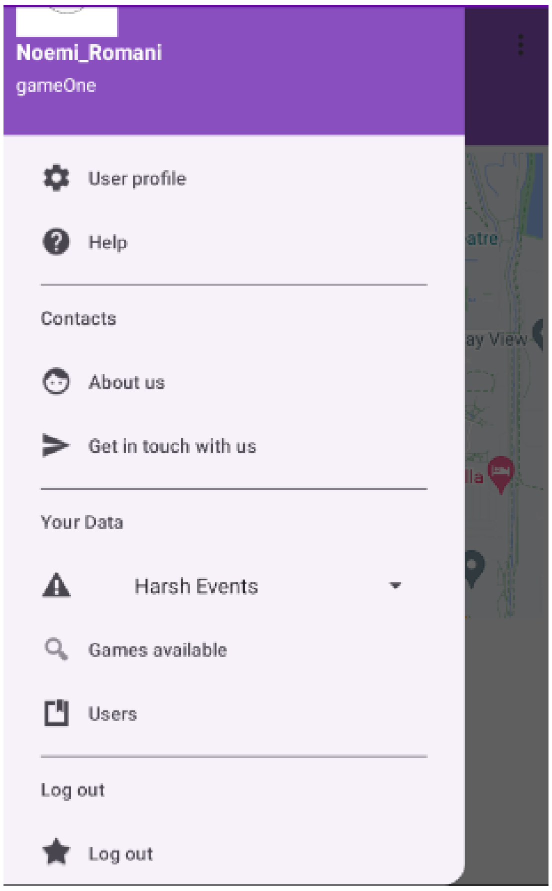


Figure 4.9: Drawer menu layout

trips.xml layout file will become part of the layout drawer via the *<include>* tag.

At the bottom of the file, there is the node *<com.google.android.material.navigation.NavigationView>*, which defines the actual layout of the Navigation Drawer component. Inside there are two items: *headerLayout*, with the associated layout describing the header menu, and *menu* with the associated drawer menu describing the options that can be selected from the Navigation drawer.

- In the Activity's java file (*TripsActivity.Class*), inside the *onCreate* method, at first is possible to find the initializations regarding the ViewPager, the TabLayout and the Toolbar. Immediately afterwards, it will be time to manage the new elements: the *DrawerLayout* [35] and the *NavigationView*

[36], introduced with the Design Support Library. Both components will, like all widgets, be found using the *findViewById* method. The *DrawerLayout* will be used to physically operate the panel, while the *NavigationView* will allow to manage the commands inserted inside it. The *NavigationView* will therefore be populated with a series of items, defined as if it were an Options Menu.

Each item in the *NavigationView* internal menu has an *id* that identifies it. The selection of each individual item will be processed by the *onNavigationItemSelected* method, belonging to a listener of type *OnNavigationItemSelectedListener*. This will be connected, again in *onCreate*, with the *NavigationView* and will thus become the official manager of the items contained therein.

In the app developed in this project, the selectable items in the Navigation drawer menu are those shown in the Activity Diagram (4.2.3).

The selection of one of them will activate the opening of a dialog window, and will physically close the "drawer". This last operation will be requested directly from the *DrawerLayout* via its *closeDrawers* method.

About the dialog window, these are small interfaces that can be customized as desired, but also exist in standard versions, which open when necessary (when the specific menu item is selected) and which the user closes at the end of their use. In the developed app, standard dialog windows have been adopted, *ProgressDialog*, which shows a progress bar and the *AlertDialog*, specialized in showing a message with a maximum number of three buttons to allow the user to respond, which have been implemented to customize the content of Dialog Fragments. In fact, to manage the dialogue windows, *DialogFragment* [37] elements were used: these are fragments that appear "floating" above the Activity and which manage a Dialog object within them.

The opening of the menu drawer is enabled via an icon from the application *Toolbar*, as indicated by the *onOptionItemSelected* method and the insertion of the lines of code relating to *getSupportActionBar*.

The activity diagram is represented in Figure 4.10. As shown, from the Navigation drawer menu, the user has the option to access his personal profile, to read information describing the app's logic and the Gamification mechanism, to get information for entering in touch with the application's managers, to check personal relevant data such as the harsh events that have been detected during the trips already done, to look at the the users participating to the available games, and to look at the list of the games resulting to be open. The last item of the menu deals with the Log out functionality.

Above, in Figure 4.9, the drawer menu is illustrated.

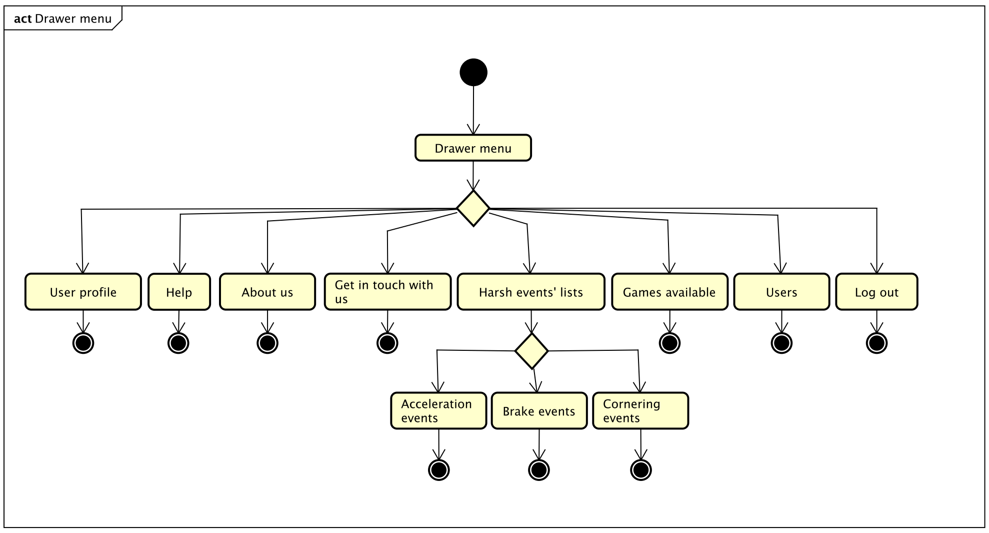


Figure 4.10: Activity Diagram: Drawer Menu

4.3 Data Layer

As explained in the introduction, this chapter moves from the UI level to the explication of the data structure. In order to describe the business logic of the application by means of the internal data structure, intended as Local Databases, the entity relationship diagram is given (4.11).

4.3.1 UML: ER Diagram

An *entity relationship (ER) diagram* is a type of flowchart that illustrates how "entities", such as people, objects, or concepts, relate to each other within a system. Also known as ERD or ER models, they use a defined set of symbols, such as rectangles, diamonds, ovals, and connecting lines, to represent the interconnection between entities, relationships, and their attributes. They mirror grammatical structure, with entities as nouns and relations as verbs. In this case, the ER diagram has been used to model and design the application's relational Database. As illustrated in the diagram, the local database is composed of the following tables: *Games, Trips, Users, Hard Acceleration, Harsh Brake, Harsh Cornering, SensorData_zValues, sensorData_xValues, GPSData.* Each of the tables is characterized by several columns, with one of them representing the Primary key; in some tables, is also possible to find columns acting as Foreign key. Is finally worth to notice the relationships between tables: in this diagram most of the connecting lines between tables represent "zero-to-many" relationships, with some cases of "one-to-one" relationships.

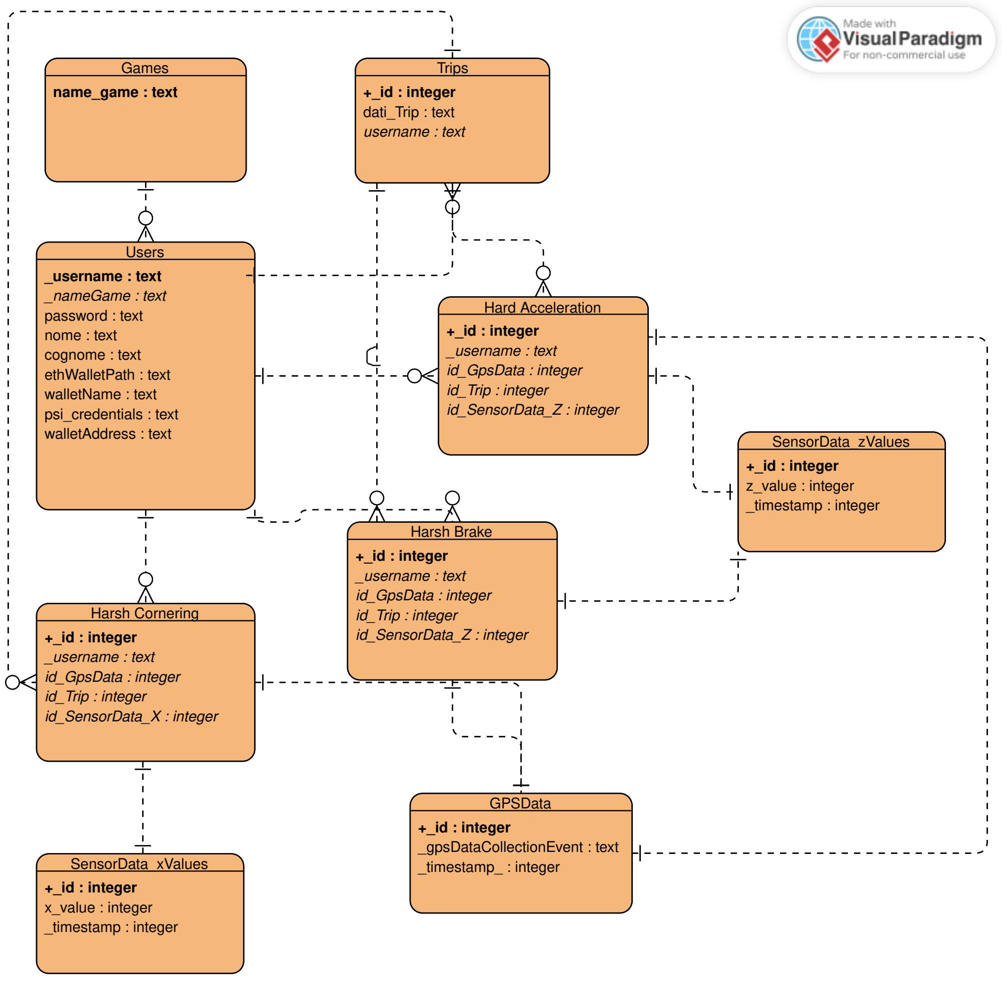


Figure 4.11: ER Diagram

Moving the focus to the programming code, the logic for the database to create tables and to insert, update, manage data is defined by the *DbHelper* and *DbManager* Classes. The *DbHelper* Class is created to facilitate the creation and configuration of a database within an Android application. It expands the *SQLiteOpenHelper* framework class, so that all the inherited abstract methods (OnCreate, OnUpgrade) are defined.

- Method *onCreate()*: this method is invoked every time the application realizes that it needs a database that does not yet exist on disk within the Android device. The database is automatically created by Android, and represented by "*db*", which is an object of the *SQLiteDatabase* Class. An object of the SQLiteDatabase Class represents indeed a reference to the application's database.

DbHelper comes into play every time it is invoked by the activity.

- Method *onUpgrade()*: in this work this method is not used, but is generally utilized to invoke upgrade of a database that exists on the device disk but is of an older version than the one requested by the software (in the application update box). For this reason, databases are characterized by a version number, passed to the constructor of the parent class (*SqliteOpenHelper*) which is invoked within the constructor of the *DbHelper* Class.

The *DbManager* Class is created to contain all the methods through which the application interacts with the database: it is the only class with database invocations by the application, therefore it contains all the command and selection queries. A *DbHelper* object is created within the class, which defines the point of creation of the internal structure of the database (the database is created automatically by Android), and it is instantiated within the constructor of the *DbManager* class. The *helper* object of the DbHelper Class is used within the application to obtain references to the database, mainly via the *helper.getReadableDatabase*, *helper.getWritableDatabase* methods.

In order to highlight the difference between data stored into the Blockchain platform with respect to those stored into the application's database, the ERD describing the data structure of the Smart Contract is presented in the Figure 4.12.

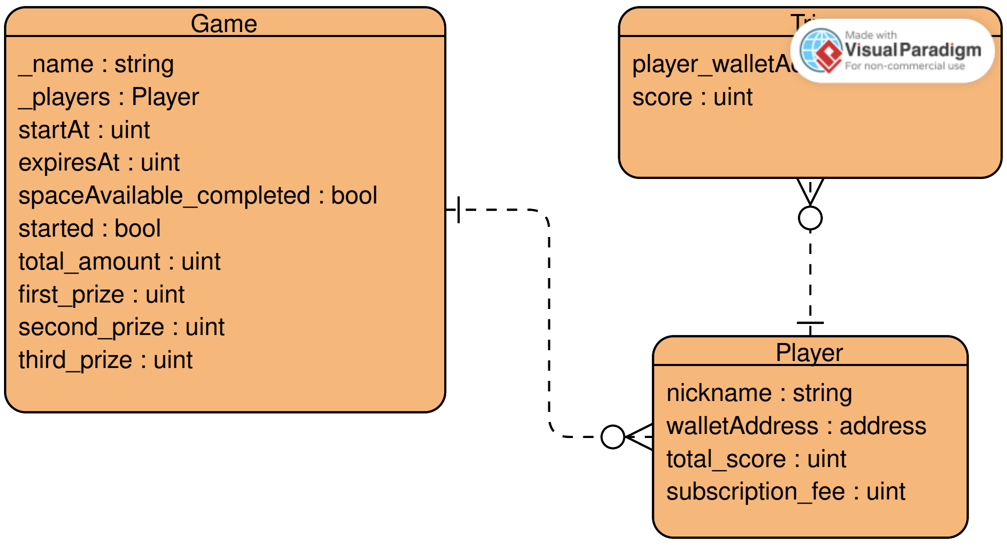


Figure 4.12: ER Diagram: Smart Contract

In this Figure 4.13, the whole structure of the Smart Contract is illustrated. This diagram has been generated directly from the Solidity UML Generator within the Remix IDE.

4.4 UML: Sequence Diagrams

The last proposed UML diagram is an interaction diagram, used to show the interactive behaviour of a system. While all the previous diagrams were static, here is possible to understand how the actors interact within the system. A sequence diagram simply depicts interaction between objects in a sequential order that is the order in which these interactions take place. The main elements of the sequence diagrams are:

- *lifelines*, vertical lines depicting an individual partecipant in a sequence diagram over time;
- *message*, represented by an arrow, representing the communication between objects;
- *activation or execution*,representing the time to complete the task.

The actors involved in the following diagrams are:

- the *User*, who is the mobile app "active" user;
- the *Mobile App System*, which allows the system to work on the functionalities building up the logic of the application;
- *services*, which are considered to manage the work of sensors;
- the *Blockchain System*, which allows to write and store data onto the Blockchain platform;
- the *Mobile App Database*, which allows the system to store and manage the application's data.

The first Sequence Diagram (4.14) describes the sequential actions starting from the press of the "Start You Trip" Button, dealing with the Ethereum Wallet generation, initialization or joining of a trip, among the functionalities defined in the IDEF0 Diagram 4.2. *The actors involved in this diagram are: User, Mobile App System, Mobile App Database and the Blockchain System.* By initializing a new game or joining an existing game, some information (name of the game, nickname of the user) are transferred to the Smart Contract leading to the creation of Game entities and a Player entities.

The second Sequence Diagram (4.15) shows the actions to be performed in the first phase of the process of detecting the drivers' behavior: from the activation of the sensors to the collection of raw data, application of filters and storage of data into the application's database. The actors involved in this diagram are: *User, Mobile App System, Mobile App Database.*

The third Sequence Diagram (4.16) illustrates the actions to be performed starting from the press of the “Stop Your Trip” Button: it deals with the phases of raw data analysis and computation of trip’s score involved in the process of detecting the drivers’ behavior, with the storage of data into the database of the application, and with the transfer of data to the Smart Contract. The actors involved in this diagram are: *User, Mobile App System, MyService and Foreground Service, Mobile App Database.*

The last Sequence Diagram (4.17) shows the mechanism of the game built on the Smart Contract. All the functionalities of the game’s structure are given, from the initialization of a new game or joining of an existing game by respecting some defined constraints, to the upload of trip’s information which are relevant to compute the actual and final ranking of the players, to the transfer of “prize money”. The actors involved in this diagram are: *User, Blockchain System, Blockchain platform.*

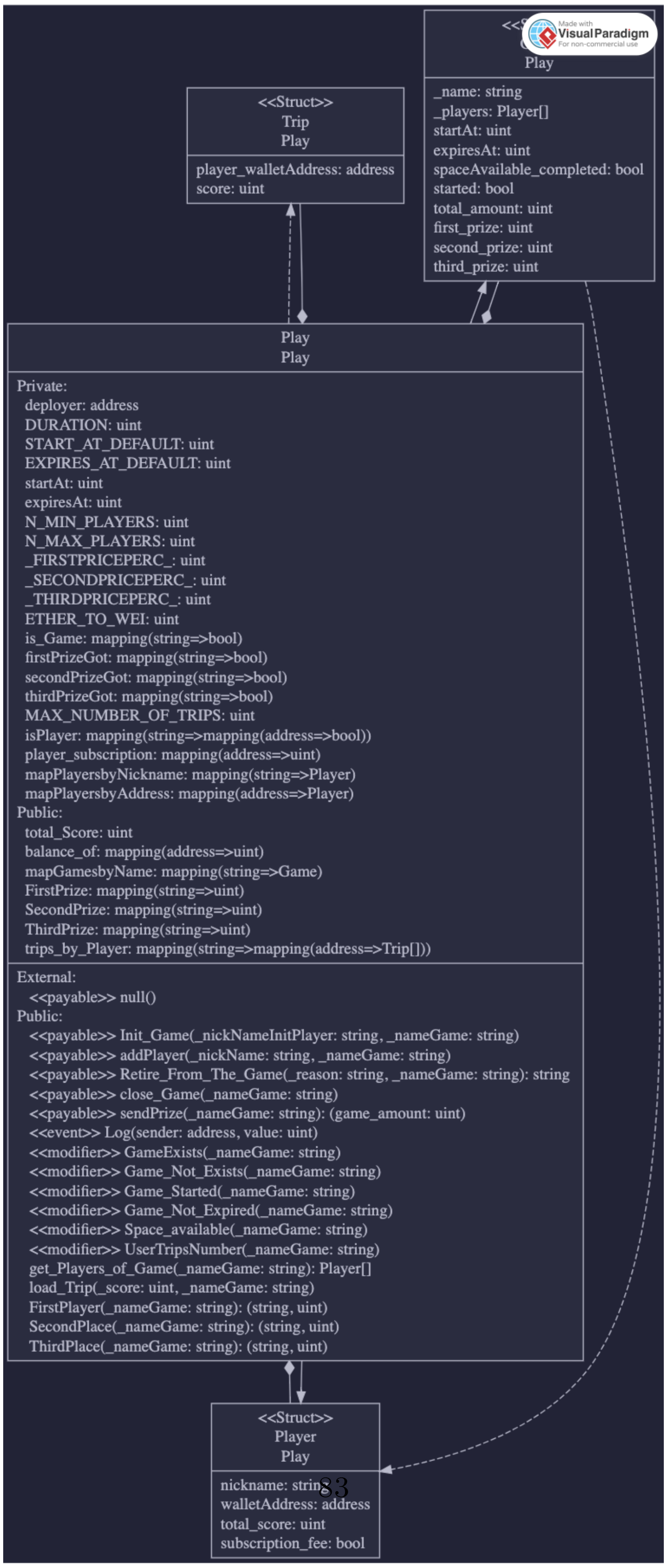
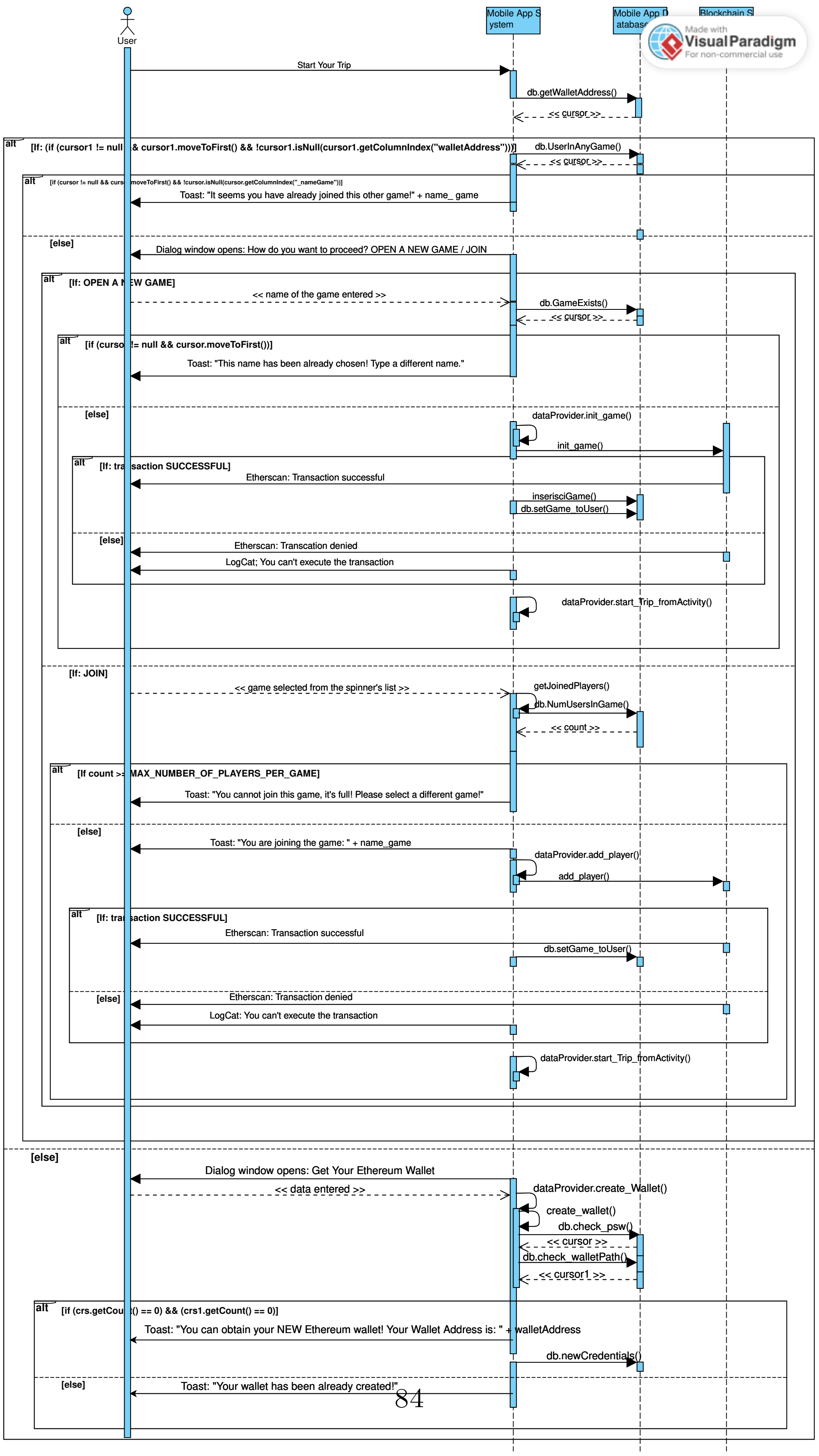

Made with Visual Paradigm For non-commercial use
<<S
Play
_name: string
_players: Player[]
startAt: uint
expiresAt: uint
spaceAvailable_completed: bool
started: bool
total_amount: uint
first_prize: uint
second_prize: uint
third_prize: uint
<<Struct>>
Trip
Play
player_walletAddress: address
score: uint
Play
Play
Private:
deployer: address
DURATION: uint
START_AT_DEFAULT: uint
EXPIRES_AT_DEFAULT: uint
startAt: uint
expiresAt: uint
N_MIN_PLAYERS: uint
N_MAX_PLAYERS: uint
FIRSTPRICEPERC: uint
SECONDPRICEPERC: uint
THIRDPRICEPERC: uint
ETHER_TO_WEI: uint
is_Game: mapping(string=>bool)
firstPrizeGot: mapping(string=>bool)
secondPrizeGot: mapping(string=>bool)
thirdPrizeGot: mapping(string=>bool)
MAX_NUMBER_OF_TRIPS: uint
isPlayer: mapping(string=>mapping(address=>bool))
player_subscription: mapping(address=>uint)
mapPlayersbyNickname: mapping(string=>Player)
mapPlayersbyAddress: mapping(address=>Player)
Public:
total_Score: uint
balance_of: mapping(address=>uint)
mapGamesbyName: mapping(string=>Game)
FirstPrize: mapping(string=>uint)
SecondPrize: mapping(string=>uint)
ThirdPrize: mapping(string=>uint)
trips_by_Player: mapping(string=>mapping(address=>Trip[]))
External:
<<payable>> null()
Public:
<<payable>> Init_Game(_nickNameInitPlayer: string, _nameGame: string)
<<payable>> addPlayer(_nickName: string, _nameGame: string)
<<payable>> Retire_From_The_Game(_reason: string, _nameGame: string): string
<<payable>> close_Game(_nameGame: string)
<<payable>> sendPrize(_nameGame: string): (game_amount: uint)
<<event>> Log(sender: address, value: uint)
<<modifier>> GameExists(_nameGame: string)
<<modifier>> Game_Not_Exists(_nameGame: string)
<<modifier>> Game_Started(_nameGame: string)
<<modifier>> Game_Not_Expired(_nameGame: string)
<<modifier>> Space_available(_nameGame: string)
<<modifier>> UserTripsNumber(_nameGame: string)
get_Players_of_Game(_nameGame: string): Player[]
load_Trip(_score: uint, _nameGame: string)
FirstPlayer(_nameGame: string): (string, uint)
SecondPlace(_nameGame: string): (string, uint)
ThirdPlace(_nameGame: string): (string, uint)
<<Struct>>
Player
Play
nickname: string
walletAddress: address
total_score: uint
subscription_fee: bool

Figure 4.13: UML Diagram Solidity


User
Mobile App System
Mobile App Database
Blockchain S
Made with Visual Paradigm For non-commercial use
Start Your Trip
db.getWalletAddress()
<< cursor >>
alt
[If: (if (cursor1 != null && cursor1.moveToFirst() && !cursor1.isNull(cursor1.getColumnIndex("walletAddress")))]
db.UserInAnyGame()
<< cursor >>
alt
[if (cursor != null && cursor.moveToFirst() && !cursor.isNull(cursor.getColumnIndex("_nameGame"))]
Toast: "It seems you have already joined this other game!" + name_ game
[else]
Dialog window opens: How do you want to proceed? OPEN A NEW GAME / JOIN
alt
[If: OPEN A NEW GAME]
<< name of the game entered >>
db.GameExists()
<< cursor >>
alt
[if (cursor != null && cursor.moveToFirst())]
Toast: "This name has been already chosen! Type a different name."
[else]
dataProvider.init_game()
init_game()
alt
[If: transaction SUCCESSFUL]
Etherscan: Transaction successful
inserisciGame()
db.setGame_toUser()
[else]
Etherscan: Transcation denied
LogCat; You can't execute the transaction
dataProvider.start_Trip_fromActivity()
[If: JOIN]
<< game selected from the spinner's list >>
getJoinedPlayers()
db.NumUsersInGame()
<< count >>
alt
[If count >= MAX_NUMBER_OF_PLAYERS_PER_GAME]
Toast: "You cannot join this game, it's full! Please select a different game!"
[else]
Toast: "You are joining the game: " + name_game
dataProvider.add_player()
add_player()
alt
[If: transaction SUCCESSFUL]
Etherscan: Transaction successful
db.setGame_toUser()
[else]
Etherscan: Transaction denied
LogCat: You can't execute the transaction
dataProvider.start_Trip_fromActivity()
[else]
Dialog window opens: Get Your Ethereum Wallet
<< data entered >>
dataProvider.create_Wallet()
create_wallet()
db.check_psw()
<< cursor >>
db.check_walletPath()
<< cursor1 >>
alt
[if (crs.getCount() == 0) && (crs1.getCount() == 0)]
Toast: "You can obtain your NEW Ethereum wallet! Your Wallet Address is: " + walletAddress
db.newCredentials()
[else]
Toast: "Your wallet has been already created!"

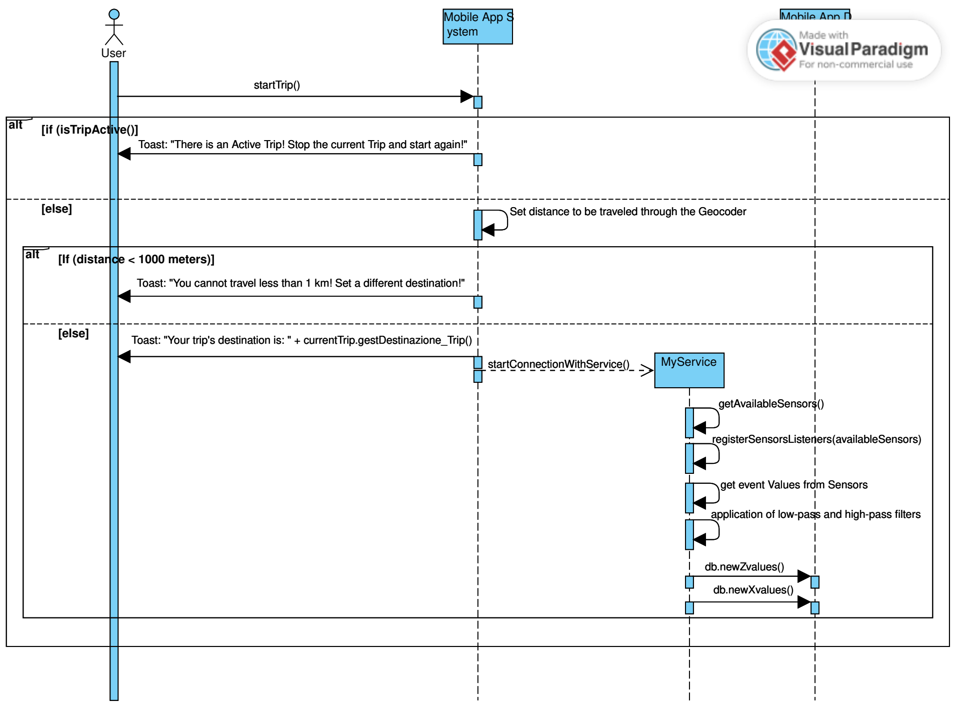
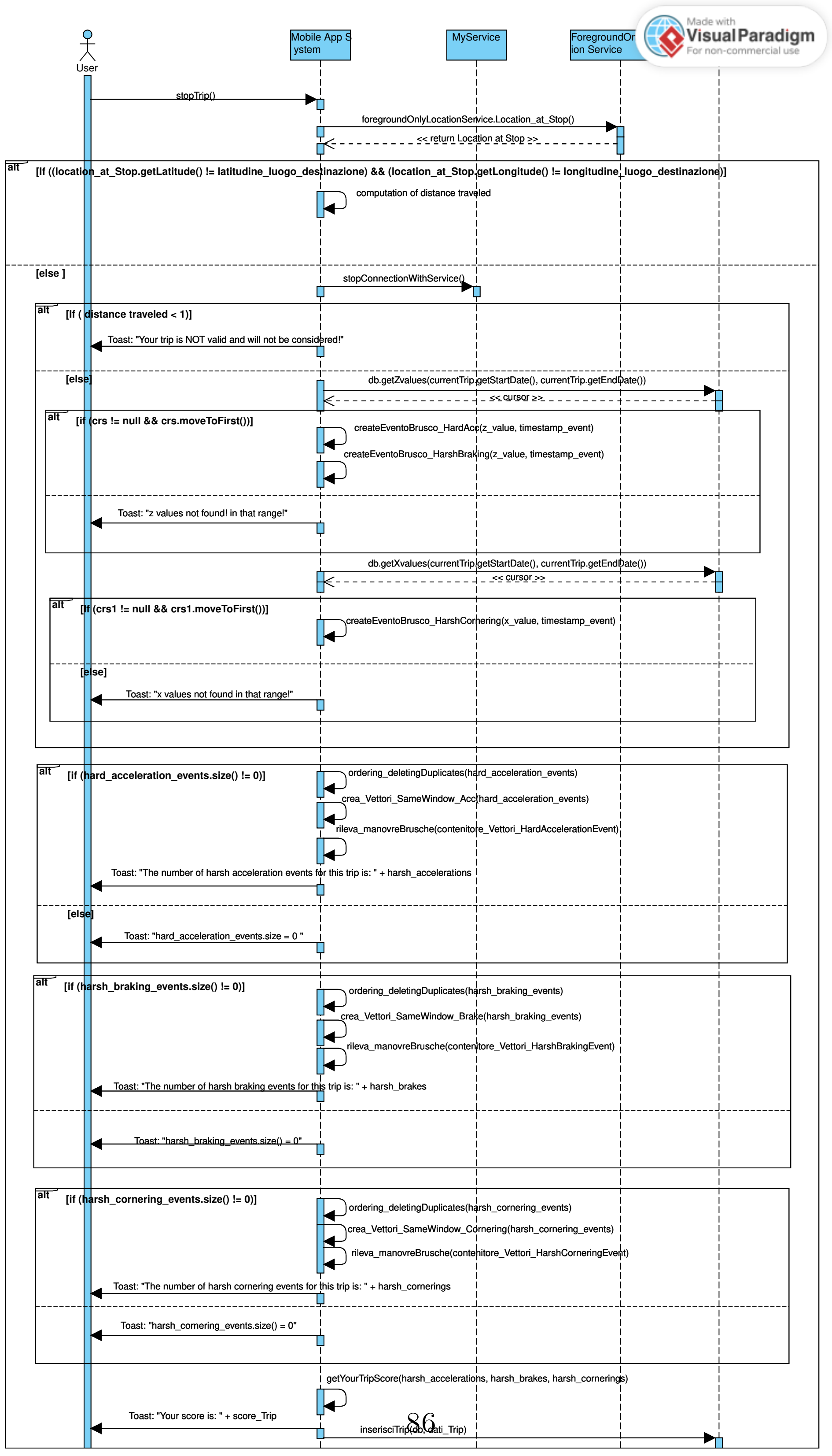

User
Mobile App S ystem
Mobile App D
startTrip()
alt
[if (isTripActive()]
Toast: "There is an Active Trip! Stop the current Trip and start again!"
[else]
Set distance to be traveled through the Geocoder
alt
[If (distance < 1000 meters)]
Toast: "You cannot travel less than 1 km! Set a different destination!"
[else]
Toast: "Your trip's destination is: " + currentTrip.gestDestinazione_Trip()
startConnectionWithService()
MyService
getAvailableSensors()
registerSensorsListeners(availableSensors)
get event Values from Sensors
application of low-pass and high-pass filters
db.newZvalues()
db.newXvalues()

Figure 4.15: Sequence Diagram


Made with Visual Paradigm For non-commercial use
User
Mobile App System
MyService
ForegroundOnlyLocation Service
stopTrip()
foregroundOnlyLocationService.Location_at_Stop()
<< return Location at Stop >>
alt
[If ((location_at_Stop.getLatitude() != latitudine_luogo_destinazione) && (location_at_Stop.getLongitude() != longitudine_luogo_destinazione)]
computation of distance traveled
[else]
stopConnectionWithService()
alt
[If (distance traveled < 1)]
Toast: "Your trip is NOT valid and will not be considered!"
[else]
db.getZvalues(currentTrip.getStartDate(), currentTrip.getEndDate())
<< cursor >>
alt
[if (crs != null && crs.moveToFirst())]
createEventoBrusco_HardAcc(z_value, timestamp_event)
createEventoBrusco_HarshBraking(z_value, timestamp_event)
Toast: "z values not found! in that range!"
db.getXvalues(currentTrip.getStartDate(), currentTrip.getEndDate())
<< cursor >>
alt
[If (crs1 != null && crs1.moveToFirst())]
createEventoBrusco_HarshCornering(x_value, timestamp_event)
[else]
Toast: "x values not found in that range!"
alt
[if (hard_acceleration_events.size() != 0)]
ordering_deletingDuplicates(hard_acceleration_events)
crea_Vettori_SameWindow_Acc(hard_acceleration_events)
rileva_manovreBrusche(contenitore_Vettori_HardAccelerationEvent)
Toast: "The number of harsh acceleration events for this trip is: " + harsh_accelerations
[else]
Toast: "hard_acceleration_events.size = 0 "
alt
[if (harsh_braking_events.size() != 0)]
ordering_deletingDuplicates(harsh_braking_events)
crea_Vettori_SameWindow_Brake(harsh_braking_events)
rileva_manovreBrusche(contenitore_Vettori_HarshBrakingEvent)
Toast: "The number of harsh braking events for this trip is: " + harsh_brakes
Toast: "harsh_braking_events.size() = 0"
alt
[if (harsh_cornering_events.size() != 0)]
ordering_deletingDuplicates(harsh_cornering_events)
crea_Vettori_SameWindow_Cornering(harsh_cornering_events)
rileva_manovreBrusche(contenitore_Vettori_HarshCorneringEvent)
Toast: "The number of harsh cornering events for this trip is: " + harsh_cornerings
Toast: "harsh_cornering_events.size() = 0"
getYourTripScore(harsh_accelerations, harsh_brakes, harsh_cornerings)
Toast: "Your score is: " + score_Trip
inserisciTrip(db, dati_Trip)

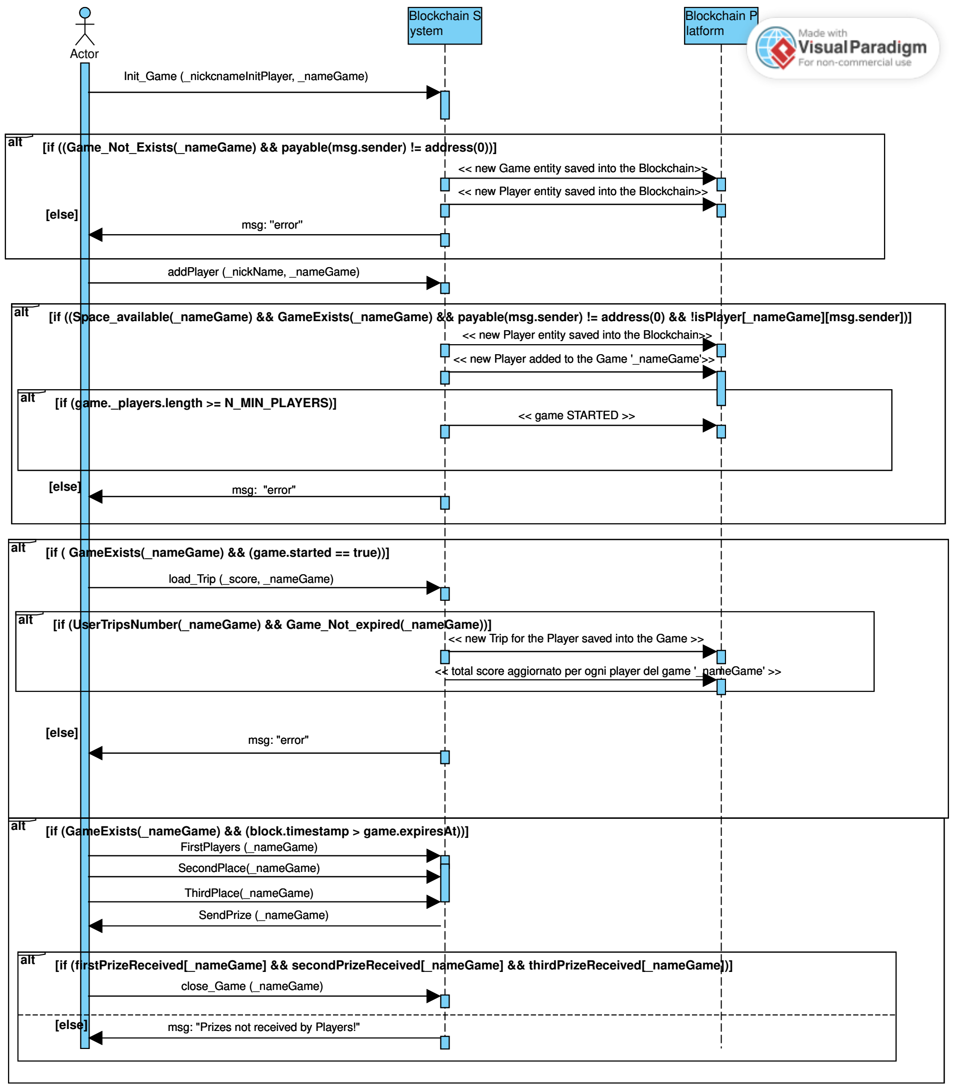

Actor
Blockchain System
Blockchain Platform
Made with Visual Paradigm For non-commercial use
Init_Game (_nickcnameInitPlayer, _nameGame)
alt
[if ((Game_Not_Exists(_nameGame) && payable(msg.sender) != address(0))]
<< new Game entity saved into the Blockchain>>
<< new Player entity saved into the Blockchain>>
[else]
msg: "error"
addPlayer (_nickName, _nameGame)
alt
[if ((Space_available(_nameGame) && GameExists(_nameGame) && payable(msg.sender) != address(0) && !isPlayer[_nameGame][msg.sender])]
<< new Player entity saved into the Blockchain>>
<< new Player added to the Game '_nameGame'>>
alt
[if (game._players.length >= N_MIN_PLAYERS)]
<< game STARTED >>
[else]
msg: "error"
alt
[if (GameExists(_nameGame) && (game.started == true))]
load_Trip (_score, _nameGame)
alt
[if (UserTripsNumber(_nameGame) && Game_Not_expired(_nameGame))]
<< new Trip for the Player saved into the Game >>
<< total score aggiornato per ogni player del game '_nameGame' >>
[else]
msg: "error"
alt
[if (GameExists(_nameGame) && (block.timestamp > game.expiresAt))]
FirstPlayers (_nameGame)
SecondPlace(_nameGame)
ThirdPlace(_nameGame)
SendPrize (_nameGame)
alt
[if (firstPrizeReceived[_nameGame] && secondPrizeReceived[_nameGame] && thirdPrizeReceived[_nameGame])]
close_Game (_nameGame)
[else]
msg: "Prizes not received by Players!"

Figure 4.17: Smart Contract Sequence Diagram

Chapter 5

Results

As introduced in the section 1, with the aim to support the necessity to overcome the limits associated to the actual PCIs and to recognize the importance of bringing drivers toward a safer driving behavior, the design goals to be achieved through the developed work were:

- data and process transparency;
- data privacy-preservation;
- data and process consistency;
- incentive for drivers toward a safe driving style.

Some of these objectives have been fully achieved, others will be certified soon by tangible results. Taking into considerations the two system's components, the mobile App and the Smart Contract, it is reasonable to separately analyze their contributions with respect to the objectives' achievement.

The objectives achieved with this thesis work can be evaluated by distinguishing the purely technical aspect, from the testing phase: if the testing phase for the Smart Contract has been fully carried out, thus certifying the consistency of the developed code and of the logic behind the Gamification mechanism, some attempts to test the Mobile App have been done, but further analysis is scheduled to follow. In the latter case, the app runs successfully on the Android Studio emulator, meaning that from a purely technical view point the developed algorithms are consistent, and run successfully in the first real driving cases, but other tests to guarantee the correct functioning when running in real conditions have to be performed.

By considering the Mobile App's contribution to the objectives' achievement, by focusing on the technical aspect, the developed programming code is consistent with:

- *Data and process transparency*: the algorithm implemented for detecting aggressive events as well as the Event-count based algorithm implemented for computing the trip's scores, are well explained to the application's users into a specific section of the drawer menu. The data considered to be relevant in the process of detecting the drivers' behavior are saved into the local database: as already said, when a harsh event is detected, the specific driver can retrieve all the sensitive data related to that event, also having the possibility to access the localization of the abrupt maneuver. The scores determined when evaluating the driving behavior of the individual trip are transferred by the Mobile App and stored on the Blockchain platform.

- *Data privacy-preservation:* referring to the internal management of data, there is a difference, as shown in Table 3.2, between data publicly accessed by all the application's users and sensitive data saved into the local database but visible only to the interested user. Referring to the data transferred by the Mobile App and written onto the Blockchain platform, only the users' nickname and the scores related to the trips will be transmitted, by preserving all the personal data and the sensitive data of the detected aggressive events.

- *Data and process consistency:* as above specified, how the app runs in real driving cases has to be further tested. It can be certified that the app runs successfully if referring to the users' registration and login, to the management of data into the local database, if referring to the effective call to the Smart Contract's methods with successful result of the transactions, and if referring to the data detection simulated by the emulator. The algorithms work successfully, the events are detected and scores computed even if not tested in real driving cases.

Moving the focus to the contribution given by the proposed Blockchain solution to the objectives' achievement, it results to be consistent with:

- *Data and process transparency:* it is an embedded characteristic of the Blockchain technology. Data which are relevant to the Gamification mechanism are public as well as the functions determining the Gamification internal logic. To address the transparency issue, it can be reasonable for Insurance Companies to design PCIs based on Smart Contracts for clarifying the evaluation protocols adopted when detecting drivers' behaviors and when determining the corresponding premiums.

- *Data privacy-preservation:* as above mentioned, the Smart Contract makes public only those data which are useful for the Gamification mechanism, and which can prove the mechanism transparency. The users' personal data as well as the driving features involved in the abrupt maneuvers are not stored on the Blockchain platform.

- *Data and process consistency:* this objective is fully achieved as the Smart Contract has been deployed on the Blockchain platform, its methods have been called by the Mobile App, and the results of the transactions are shown on Etherscan, as illustrated in Figure 5.1.
- *Incentive for drivers toward a safe driving style:* even if the drivers' incentive proposed by ICs could consist in lower insurance fees, in this thesis work this objective has been achieved by inserting drivers into a Gamification mechanism. Depending on the final ranking, the first three players will be awarded with a prize consisting in a percentage of the total amount collected by summing the subscription's fees sent to the Smart Contract by the game's players.

As regard the transparency and privacy-preservation of data, different Blockchains could be tested with the objective to individuate the solution resulting to be the most consistent with these objectives achievement: as an example, a possible solution could be a Consortium Blockchain, with the drivers being free to upload data according to their will, and with the possibility to inspect data by involving in the system model some Authorities. In case of data-cheating, some penalties could be inflicted to drivers. About the incentive toward a safer driving style, which can lead to reduced pollution levels, users may be incentivized to use their vehicles more in order to climb the ranking and achieve the highest score, which could cancel out the benefits of safe driving from an environmental perspective. To address this issue, user registration could be limited to those with electric vehicles, thus incentivizing a transition to electric vehicles, a key aspect of reducing pollutant emissions.

Finally, while measures to protect users' privacy have already been implemented in both the two components (Mobile App and Smart Contract), additional analysis and integration of security tools would provide users with an even more secure and reliable solution that does not compromise their privacy.

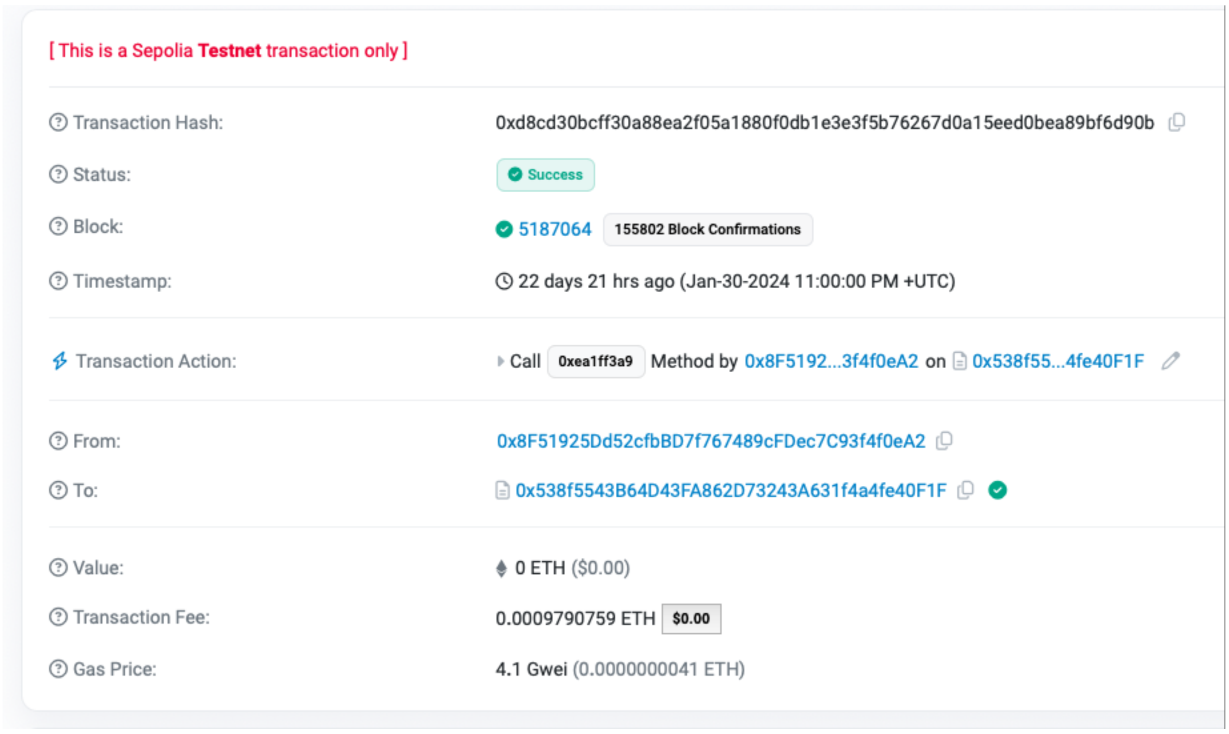
5.1 Testing the Smart Contract

To test the code developed into the Smart Contract, the following tools have been used:

- *Remix*: it is a Solidity IDE [38] for compilation/distribution using Metamask [39] [17], which allows the manual testing of the various functions.
- *Truffle*: it is a complete tool in terms of Smart Contracts' management, allowing, as already said, the distribution, compilation, interaction and testing in a dynamic way [18]. Testing is permitted thanks to the creation and subsequent execution by Truffle of particular Javascript scripts that distribute a contract with pre-estabilished parameters and call the various functions.

The Smart Contract has been deployed on the Blockchain platform by mean of a Metamask wallet address, considered to be the address of the deployer to which, in case of remaining funds on the Smart Contract at the expiration date of the specific game, funds are sent.

Here the Figure 5.1 represents the outcome of the transaction executed when the application's user called the method to initialize a new game.



[This is a Sepolia **Testnet** transaction only]

Transaction Hash:	0xd8cd30bcff30a88ea2f05a1880f0db1e3e3f5b76267d0a15eed0bea89bf6d90b
Status:	Success
Block:	5187064 155802 Block Confirmations
Timestamp:	22 days 21 hrs ago (Jan-30-2024 11:00:00 PM +UTC)
Transaction Action:	Call 0xea1ff3a9 Method by 0x8F5192...3f4f0eA2 on 0x538f55...4fe40F1F
From:	0x8F51925Dd52cfbBD7f767489cFDec7C93f4f0eA2
To:	0x538f5543B64D43FA862D73243A631f4a4fe40F1F
Value:	0 ETH ($0.00)
Transaction Fee:	0.0009790759 ETH $0.00
Gas Price:	4.1 Gwei (0.0000000041 ETH)

Figure 5.1: Etherscan: successful transaction executed

Going deeply into the analysis of the achieved results, a cost analysis for deploying the Smart Contract and executing its main functions is proposed.

As stated in Section 2.1.4, Blockchains leverage consensus algorithms to ensure decentralization, reliability, and data security by allowing nodes in the network to agree and validate transactions. There are different types of consensus algorithms based on various strategies, but they all require computational effort, which must be paid for. Storing information or running code on a Blockchain incurs a cost that varies depending on the Blockchain, network congestion, and the value of the cryptocurrency used for payment. Currently, these parameters are highly variable, resulting in significant volatility in the cost of executing transactions over time.

Among the various public Blockchains available, the Ethereum one has been chosen because it results to be the most commonly used Blockchain for developing decentralized applications and programming smart contracts, despite not being the most cost-effective. Smart contracts on Ethereum are executed through the Ethereum Virtual Machine (EVM), which is a virtual machine shared among all nodes in the network. The cost of executing functions is measured in Gas Units, with the price per Gas Unit being affected by network congestion and cryptocurrency value. Additionally, users can add a higher or lower tip to prioritize function execution. At the time of this work writing, a single unit of gas on Ethereum had an average base cost of 21 gwei, equivalent to approximately $0,71.

Table 5.1 displays the gas units required to execute the primary functions of the Smart Contract and the corresponding dollar value calculated by multiplying the gas units by the base cost of each unit. The calculation reveals that the most expensive function is *SendPrize*, with a cost of $24.28, which is relatively high. The focus was on the feasibility and operation of the Smart Contract. Therefore, significant emphasis on optimizing the code has not placed, which could potentially reduce the number of gas units required for execution.

Table 5.1: Cost analysis of the main functions on Ethereum and Polygon

Function	Gas units	Ethereum (ETH)	USD ($)	Polygon (MATIC)	USD ($)
Init_game	305160	0,00640	21,55	0,01037	0,01
Add_player	238358	0,00500	16,84	0,00810	0,00
Load_trip	136426	0,00286	9,63	0,00286	0,00
SendPrize	343420	0,00721	24,28	0,01167	0,01
Retire_From_The_Game	95906	0,00201	6,77	0,00326	0,00

Despite the optimizations, Ethereum remains an expensive solution. To reduce costs, the cost of the Smart Contract has been checked on a cheaper public Blockchain. Polygon was chosen, which is also based on EVM, allowing to use the same Smart contract without any changes from Ethereum. The units of gas required to execute the functions remain unchanged since they are always executed on EVM. The cost per unit of gas has changed, and as of the time of writing this paper, it is equivalent to 34 GWei, which corresponds to $2.10. Table 5.1 presents the same calculation on both Ethereum and Polygon, and it shows that the most expensive functions are *SendPrize* and *Init_game*, with a total cost of $0.01. Thus, regardless of optimizations, the Blockchain solution proposed by this work would not only be functional but also cheap and exploitable if the Smart Contract were deployed on a Polygon-like Blockchain.

Chapter 6

Conclusions and future works

Lack of transparency is one of the main concerns that drivers associate to the Personalized Car Insurances currently proposed by the Insurance Companies, and aggressive driving maneuvers are the leading cause of traffic incidents. Devising system models based on transparent processes and algorithms for both detecting and classifying the drivers' behavior becomes crucial. With the aim to go beyond the current lack of transparency of the PCIs proposed by the Insurance Companies, where neither the evaluation protocol nor the system of detecting driving-related data is clear, and with the aim to bring drivers toward a safer driving style, a decentralized solution with a system model based on two main components was designed. The objective of this work was to develop and test a prototype platform (Mobile App, Smart Contract, and DB) to demonstrate the feasibility and its benefits to users. More specifically, if considering the Mobile App, one of the two system components, a clear process for detecting the drivers' behavior was designed and developed through the programming code. Starting from raw data acquired through sensors installed on a smartphone, then cleaned up through the application of filters for removing noises, the core of the process consisted in the design and implementation of two robust algorithms: a first algorithm, implemented for detecting harsh events, based on the *"simple- threshold"* algorithm described in the section 3.1, and on reference values taken from the table shown in Figure 3.6 according to which threshold values of the driving features have been determined. A second algorithm, that was instead developed for the score's computation associated to the driver's behavior in each trip. This algorithm is Event-count based, meaning that it hardly depends on the number of harsh events detected per each driving feature. In this work, the basic driving features were considered: acceleration, brake, turning, but future works could consider to expand them, i.e, u-turn event, overtaking maneuver; the programming code related to this driving features was developed but not yet implemented in the programming code. About the process of detecting the harsh events, a different and more robust algorithm is planned to be inspected and adopted instead of the simple-threshold (RSS threshold-method, jerk method). Furthermore, future works also plan to test the application in a larger number of real driving cases in order to test the scalability of the solution an to increase the scheme's security. Referring to the contribution given by the Blockchain solution proposed in this work, the incentive for drivers to follow a safer driving style lied in a Gamification mechanism, which is an innovation in this context: anyway, the Gamification mechanism has to be considered as a starting point for future Insurance Companies' solutions. Future works plan to consider the exploitation of the the Blockchain technology to design transparent protocols defining the PCIs' rules, to make comparison between different Blockchain models and different system's models in order to individuate the Blockchain solution that most fills with preservation and transparency of data. The Gamification mechanism, instead, is planned to be exploited to implement further premiums' percentage reductions, or to reward the safest players with different gadgets. Additionally, the limitations outlined in Section 5 must be addressed. In future works, it is planned to prioritize the security of both the Smart Contract and the Mobile Application to create a platform that is resilient to cyber-attacks and provides reliability to users. In terms of cost reduction, it is planned to test and compare the implementation on various public Blockchains to identify the ones that offer the necessary technical features and cost-effectiveness.

Bibliography

[1] V. Gatteschi, A. Cannavò, F. Lamberti, L. Morra, and P. Montuschi. *Comparing Algorithms for Aggressive Driving Event Detection Based on Vehicle Motion Data.* 2021 (cit. on pp. 1, 2, 32, 37, 38).

[2] C. Huang, W. Wang, D. Liu, R. Lu, and X. Shen. *Blockchain-Assisted Personalized Car Insurance With Privacy Preservation and Fraud Resistance.* 2023 (cit. on pp. 2, 14).

[3] R. Bianchi G. Chiap J. Ranalli. *Blockchain: Tecnologia e applicazioni per il business: Tutto ciò che serve per entrare nella nuova rivoluzione digitale, Hoepli.* 2019. ISBN: 978-8820390075 (cit. on pp. 5, 6, 10, 18, 21).

[4] *Italy joins European partnership on blockchain supporting the delivery of cross-border digital public services.* URL: `https://digital-strategy.ec.europa.eu/en/news/italy-joins-european-partnership-blockchain-supporting-delivery-cross-border-digital-public` (cit. on p. 12).

[5] *Banking on Blockchain.* URL: `https://www.accenture.com/us-en/services/blockchain/blockchain-financial-services-infrastructure` (cit. on p. 13).

[6] *Blockchains overview: Ripple.* URL: `https://www.geeksacademy.it/articolo-212/blockchains-overview-ripple/` (cit. on p. 14).

[7] Szabo N. *Smart Contracts.* URL: `https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html/` (cit. on p. 15).

[8] *A comparison of blockchain network latencies.* URL: `https://medium.com/klaytn/a-comparison-of-blockchain-network-latencies-7508509b8460` (cit. on p. 19).

[9] *Ethereum Avg. Transaction Fee historical chart.* URL: `https://bitinfocharts.com/comparison/bitcoin-transactionfees.html#3y` (cit. on p. 19).

[10] F. Montagna M. Cantamessa. *Management of Innovation and Product Development: Integrating Business and Technological Perspectives, Springer.* 2016. ISBN: 978-1447167228 (cit. on pp. 21, 23).

[11] *Ethereum Transactions Per Day.* URL: https://ycharts.com/indicators/ethereum_transactions_per_day (cit. on p. 21).

[12] *Blockchain technology cloud market size worldwide in 2021, with a forecast for 2030.* URL: https://www.statista.com/statistics/1319369/global-blockchain-technology-market-size/ (cit. on p. 23).

[13] *Ecco perchè le assicurazioni devono investire nella Gamification.* URL: https://www.insurzine.com/2017/03/07/ecco-perche-le-compagnie-devono-investire-nella-gamification/ (cit. on p. 26).

[14] *Assicurazioni Generali sperimenta la gamification con MyGame.* URL: https://www.gameifications.com/assicurazioni-generali-sperimenta-la-gamification-con-mygame/ (cit. on p. 26).

[15] *The Vitality Group Motivates with Data.* URL: https://www.vitalitygroup.com/insights/the-vitality-group-motivates-with-data/ (cit. on p. 27).

[16] *Android Studio.* URL: https://developer.android.com/studio (cit. on p. 28).

[17] *Remix.* URL: https://remix.ethereum.org/ (cit. on pp. 28, 90).

[18] *Truffle.* URL: https://archive.trufflesuite.com (cit. on pp. 28, 90).

[19] *Web3py documentation.* URL: https://web3py.readthedocs.io/en/stable/ (cit. on p. 29).

[20] *How to implement Ethereum blockchain in Android using Web3j.* URL: https://boemo1mmopelwa.medium.com/implementing-etherium-blockchain-in-android-with-web3j-485ea0747088 (cit. on pp. 29, 56).

[21] *Etherscan.* URL: https://etherscan.io (cit. on p. 29).

[22] *Ethereum.* URL: https://it.wikipedia.org/wiki/Ethereum (cit. on p. 29).

[23] *Sensors Overview: sensor coordinate system.* URL: https://developer.android.com/develop/sensors-and-location/sensors/sensors_overview?hl=it (cit. on p. 32).

[24] *Motion sensors: Work with raw data: Use the accelerometer.* URL: https://developer.android.com/develop/sensors-and-location/sensors/sensors_motion?hl=it (cit. on p. 35).

[25] *Bound service overview.* URL: https://developer.android.com/develop/background-work/services/bound-services#java (cit. on p. 47).

[26] *Request Location Update.* URL: https://developer.android.com/develop/sensors-and-location/location/request-updates?hl=it (cit. on p. 47).

[27] *Foreground services.* URL: `https://developer.android.com/develop/background-work/services/foreground-services` (cit. on p. 47).

[28] *Notifications overview.* URL: `https://developer.android.com/develop/ui/views/notifications` (cit. on p. 50).

[29] *Guide to app architecture: UI layer page.* URL: `https://developer.android.com/topic/architecture/ui-layer` (cit. on pp. 64, 66).

[30] *Guide to app architecture: data layer page.* URL: `https://developer.android.com/topic/architecture/data-layer` (cit. on p. 64).

[31] *Fragments: create a fragment.* URL: `https://developer.android.com/guide/fragments/create` (cit. on p. 69).

[32] *ViewPager.* URL: `https://developer.android.com/reference/androidx/viewpager/widget/ViewPager` (cit. on pp. 72, 73).

[33] *PagerAdapter.* URL: `https://developer.android.com/reference/androidx/viewpager/widget/PagerAdapter` (cit. on p. 72).

[34] *TabLayout.* URL: `https://developer.android.com/reference/com/google/android/material/tabs/TabLayout` (cit. on p. 73).

[35] *DrawerLayout.* URL: `https://developer.android.com/reference/androidx/drawerlayout/widget/DrawerLayout` (cit. on p. 76).

[36] *NavigationView.* URL: `https://developer.android.com/reference/com/google/android/material/navigation/NavigationView` (cit. on p. 77).

[37] *Display dialogs with DialogFragment.* URL: `https://developer.android.com/guide/fragments/dialogs` (cit. on p. 77).

[38] *Solidity Documentation.* URL: `https://docs.soliditylang.org/en/v0.8.3/` (cit. on p. 90).

[39] *Metamask.* URL: `https://metamask.io` (cit. on p. 90).