ABCDE -- Agile Block Chain Dapp Engineering

Preprint · December 2019

CITATIONS
0

READS
12,946

3 authors:

Lodovica Marchesi
Università degli studi di Cagliari
18 PUBLICATIONS **252** CITATIONS
SEE PROFILE

Michele Marchesi
Università degli studi di Cagliari
328 PUBLICATIONS **9,681** CITATIONS
SEE PROFILE

Roberto Tonelli
Università degli studi di Cagliari
166 PUBLICATIONS **2,458** CITATIONS
SEE PROFILE

ABCDE –Agile Block Chain Dapp Engineering

Lodovica Marchesi, Michele Marchesi, Roberto Tonelli

DMI, University of Cagliari

Abstract

Cryptocurrencies and their foundation technology, the Blockchain, are reshaping finance and economics, allowing a decentralized approach enabling trusted applications with no trusted counterpart. More recently, the Blockchain and the programs running on it, called Smart Contracts, are also finding more and more applications in all fields requiring trust and sound certifications. Some people have come to the point of saying that the "Blockchain revolution" can be compared to that of the Internet and the Web in their early days. As a result, all software development revolving around the Blockchain technology is growing at a staggering rate. The feeling of many software engineers about such huge interest in Blockchain technologies is that of unruled and hurried software development, a sort of competition on a first-come-first-served basis which does not assure neither software quality, nor that the basic concepts of software engineering are taken into account. This paper tries to cope with this issue, proposing a software development process to gather the requirement, analyze, design, develop, test and deploy Blockchain applications. The process is based on several Agile practices, such as User Stories and iterative and incremental development based on them. However, it makes also use of more formal notations, such as some UML diagrams describing the design of the system, with additions to represent specific concepts found in Blockchain development. The method is described in good detail, and an example is given to show how it works.

Keywords: Blockchain, Smart Contracts, Blockchain-oriented software engineering, UML, dApp design

1. Introduction

The so-called "decentralized applications", or "dApps", are one of the main new trends of software development. dApps typically run on a blockchain, the technology originally introduced to manage the Bitcoin digital currency [1]. Blockchain software runs in a network of peer-to-peer nodes, so it is naturally decentralized, redundant and transparent. A few years after the introduction of Bitcoin in 2009, developers and managers realized that a blockchain can be also the ideal environment for a decentralized computer. This led to the introduction of Ethereum blockchain, a network whose nodes are also able to run Turing-complete programs [2], called "Smart Contracts" (SCs) following and idea of Nick Szabo [3]. SCs are general computer programs, though with some specific features. The main idea behind them is that they can be used for the automated enforcement of contractual obligations, without having to trust a central authority, and without space and time constraints. So, there is a huge wave of interest in SCs and applications of the blockchain, especially in the financial realm. Some authors even said that *"we should think about the Blockchain as another class of thing like the Internet [...]"* [4] and that the *"wide adoption of Blockchain technology has the potential of reshaping the current financial services technical infrastructure."* [5].

This interest led to a huge amount of money flooding into Blockchain ventures. During 2017, this steady inflow of money, paired with the limited amount of available digital money – a feature that most digital currencies have by design – made the price of Bitcoin and other digital currencies spike at the end of the year. This peak was mainly ignited by the Initial Coin Offers (ICO) phenomenon, where a startup publishes a white paper describing their idea, and gathers digital money by issuing a token, which is a second level digital currency, managed by a SC [6]. This

Email address: lodo.marchesi@gmail.com, marchesi@unica.it, roberto.tonelli@dsf.unica.it (Lodovica Marchesi, Michele Marchesi, Roberto Tonelli)

token can then be immediately traded on an exchange (a pseudo bank running on the Web, which allows to exchange digital currencies against traditional money, or against other digital currencies). The enthusiasm for this new idea, the ever increasing prices and profits, and the *fear of missing out* (FOMO) led to many billions USD pouring into tokens. At the beginning of 2018 the bubble deflated, with the global capitalization of digital currencies going from more than 800 billion USD on January 7, 2018, to about 115 billion USD on February 2019. During 2019, however, a renewed interest to digital currencies lead their global market cap back to about 200 billion USD as of September. Huge inflows of money and venture capital were also poured into blockchain initiatives not linked to digital currencies. These are the so called "permissioned" blockchains, or distributed ledgers (DL), intended to be run by a set of nodes chosen by invitation.

All the initiatives behind blockchain technology – new digital currencies with their own blockchain, exchanges and other Web-based ventures using digital money, ICO startups, applications running on permissioned blockchains or DLs – are based on developing a new software system. This often led to a run to be the first on the market, as always happens with new technology waves, with quick application development, neglecting good development practices, and often even basic testing and security assessment.

Some big disasters quickly followed, with a total of literally billions of USD (at least at the nominal exchange rate) of digital currencies stolen or lost. Several exchanges were hacked, since the beginning of Bitcoin trading, for a total of various billion equivalent USD (for a list up to 2017, see [7]). Also, SCs were often exploited, taking advantage of their novelty and of the hurried software development [8], [9].

Overall, the scenario looks that of a competition on a first-come-first-served basis, where the basic principles and practices of software engineering (SE) are not taken into account. The quality of the resulting software is accordingly compromised.

It is well known that, to develop a reliable and maintainable software system, one needs to follow an explicit development process, and use sound SE practices. Among the latters, in the context of blockchain development, we stress the importance of requirement elicitation, system design, specific notations and security assessment. In essence, we need blockchain-oriented software engineering (BOSE) [10].

In this paper we present a development process for applications based on Smart Contracts running on a blockchain, which are usually called "dApps" (decentralized applications). The process covers all the standard phases of software life cycle: requirement elicitation, design, implementation, security assessment and testing, and ongoing maintenance. We call the process "ABCDE", Agile Block Chain Dapp Engineering. ABCDE is an agile software development process, meaning that is follows the principles of Agile Manifesto [11]. However, we had to complement the agile process with a more formal approach, using UML diagrams with a specific notation for SCs, and a specific checklist for security assessment.

The first question we had to answer regarding ABCDE is "why a new process"? Why not use an existing process, waterfall or agile, for dApp development? The answer to this question stems from the observation that a SC is very peculiar software. It runs on all the nodes hosting a Blockchain, and its execution has the strong constraint that all outputs and state changes resulting from SC execution must be the same in all nodes. Consequently, a SC is strictly forbidden to access the external word – it can answer to external messages belonging to its public interface, and can send messages to other SCs running on the same blockchain; no other kind of interaction with the external world is allowed. This fact implies that any dApp is intrinsically divided in two subsystems – the SCs running on a blockchain, and the applications allowing users and devices to interact with the SCs. Another specificity of SC realm is the need to introduce new concepts with respect to traditional programming, like those of "address" to refer to SCs; of signed "transaction" to send a message starting from a given address; of "GAS" needed to run a SC; of digital money owned by, and transferred between, SCs; of "oracle", a SC able to provide data coming from the external world without violating the constraint described above. Moreover, referring to Solidity, the programming language of Ethereum, which is presently the most used blockchain actually running SCs, there are further specific concepts, such as those of "modifier" (a boolean function acting as a guard to the execution of another function), of "library contract", and there are constraints on the use of typical structures of object oriented programming languages. For instance, inheritance is limited, and the available collections are just arrays and mappings. Finally, two papers by Chakraborty, Bosu et al. [12] [13], present the results of a survey among blockchain developers. They found that the prevalent opinion is that blockchain development is different from traditional one, due to the strict and non-conventional security and reliability requirements, and to other unique characteristics of the dApp development domain, such as immutability, difficulty in upgrading the software, and so on. More information on this survey is presented in Sec. 3.

This specificity led us to conclude that a new method is needed for dApp development. In fact, ABCDE is not entirely new, but it is a significant extension to classical agile methods, such as Scrum [14]. With respect to Scrum, ABCDE does not only describe how the development should be managed, but also introduces specific practices such as the use of modified UML diagrams to describe SCs, and checklists for security assessment. Other agile practices, such as simplicity, test driven development [15], refactoring [16], collective code ownership, pair programming, are encouraged if the team feels they are useful, but are not prescribed by ABCDE.

The main characteristic of ABCDE is the split of dApp development in two flows which are carried on concurrently, after a common start. The first flow is the specification and development of the SCs. The second flow is about the development of the software applications which allow external actors to interact with the SCs.

In this paper, we also introduce a notation augmenting some UML diagrams (Use Case, Sequence, and Class diagrams) to account for SC specificity in the context of Solidity language. The dApp design when other languages to develop SCs are used can be represented with different UML extensions. In this paper, we limit ourselves to Solidity.

The proposed ABCDE methods has been tested on some real dApp development projects, carried on at our department, and at some firms we are consultant of.

The remainder of this paper is organized as follows. In Section 2 we describe the architecture of a dApp, and introduce the specific issues and practices needed for dApp development. In Section 3 we present the related work in the same, or similar fields. Section 4 describes the proposed ABCDE process in every detail, including the modifications of some UML diagrams to cope with Solidity concepts, security assessment and what is needed to extend the notation to other languages for SC development. A simplified example, drawn from a real case, is presented in Section 5, together with reporting on actual uses of ABCDE. Finally, Section 6 presents the conclusions and future work ideas.

2. Background

2.1. Decentralized applications

We define as dApp (decentralized application) a software system that uses distributed ledger technology (DLT), typically a blockchain[1], as a central hub to store and exchange information, through Smart Contracts (SCs). Note that it is not a blockchain software able to manage a new cryptocurrency or other applications – that is, software enabling blockchain nodes, which needs different kinds of of development practices, not the subject of this work.

A blockchain is a distributed data structure – managed by a set of connected nodes – characterized by the following elements:

- it is redundant (each node holds a copy of the blockchain);
- it is append-only – once written, the information cannot be changed or deleted;
- the blockchain state is changed by sending *transactions* to the network – in *public blockchains*, everyone can send a transaction;
- all transactions are checked by the reached nodes; invalid ones are ignored;
- the valid transactions are typically recorded in sequentially ordered blocks – hence the name "blockchain" – whose creation is managed by a consensus algorithm among the nodes;
- all transactions are sent from an unique address, which is in turn computed from a public key. Only the owner of the private key associated with it can sign the transactions coming from this address using asymmetric cryptography, validating them;
- if the blockchain is able to execute SCs, a transaction can create a SC, or execute one of its public functions; in this case, the function is executed by all nodes, when the transaction is evaluated – the execution of the program *is* the execution of the transaction.

[1] From now on, we'll use the terms "DLT" and "blockchain" interchangeably.

A dApp is usually composed of SCs deployed on a blockchain, and of software able to create and send transactions to them. This software usually provides a user interface, running on a PC, or on a mobile device. Additional information could be stored on a server, and further business logic could be executed on this.

Most present real applications of dApps and SCs are intended for the management of digital currencies or tokens, that have a true monetary value. The use of dApps has been introduced also for other scopes, like notarization of information, identity management, voting, games and betting, goods provenance certification, and many others [17].

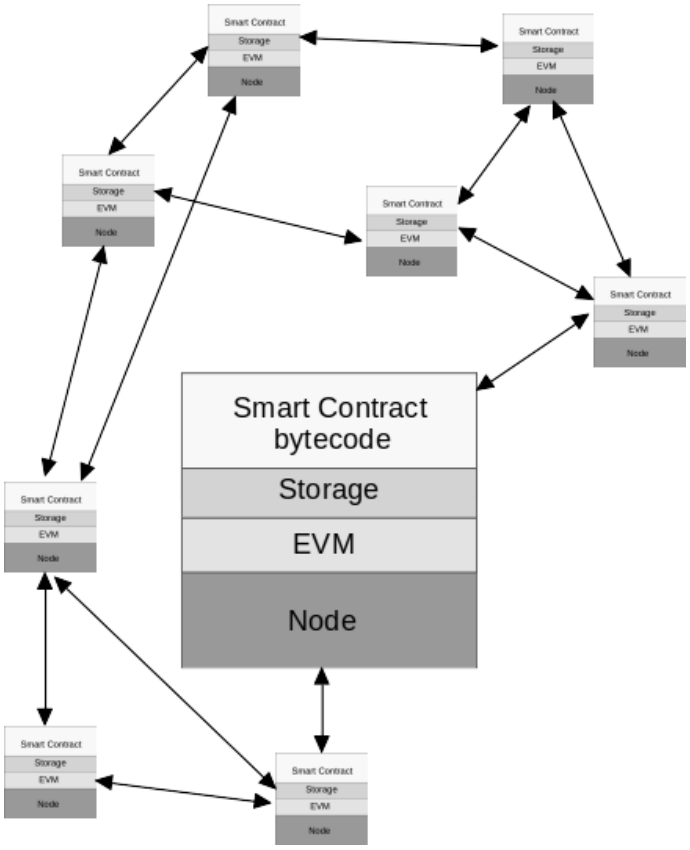

Figure 1: The Ethereum Blockchain running a SC. The same SC bytecode is executed by each node.

In this paper, we will use as a reference Ethereum, whose SCs were the first to exhibit a Turing-complete capability, and is presently the most used blockchain to develop SCs [18]. Just to quote some figures, as of October 1, 2019 there were 1533 out of 1720 active tokens[2] managed by Ethereum SCs, worth more than 8 billion USD at current prices [19]. Moreover, as of November 2019, 2654 dApps were running on Ethereum, out of a total of 3169 surveyed dApps [20]. These figures are about public blockchains. Data on dApps running on permissioned blockchains are more difficult to find, but Ethereum is very popular also for this kind of dApps. Open source DLTs such as Hyperledger and Corda are also widely used.

The Ethereum Virtual Machine (EVM), able to execute SC Ethereum bytecode, runs on all nodes of the Ethereum blockchain [21]. In practice, the SCs are written in high-level languages (HLL). Nowadays, the most popular HLL for Ethereum is called Solidity. Other languages, such as Flint and Vyper can be used, but their adoption is still far behind Solidity. Fig. 1 shows a sketch of the Ethereum blockchain, with the architecture of its nodes running a SC. The original Ethereum software running the node is written in C++ or Go language. A compatible implementation written in Rust language (Parity) is also available.

[2]We define as "active" a token whose market cap is over 100,000 USD.

As written in the Introduction, SCs run in an isolated environment. The results of their execution must be the same whatever node they run in; consequently, they cannot get information from the external world (which mutates with time), and cannot initiate a computation autonomously (for instance at given times). SCs can only access and change their state, and send messages to other SCs. The state of a SC is permanently stored in the blockchain, using *storage* variables. Another SC specificity is their immutability. Once a SC is deployed, it is in the blockchain forever – it cannot be modified or erased, though it can be forever disabled.

SCs are created by special transactions. Creating a SC and changing its state costs units of "GAS", which must be paid in Ether (the digital currency of the Ethereum Blockchain). Each SC has a unique Ethereum address, that is used to send messages to it. In Solidity, a SC can inherit from other SCs; it has a public interface, that is a set of functions that can be called through a transaction. The call of a public function of a SC is called a "message". Sending a message to a SC can be performed either by posting a transaction coming from an address, or by executing code of the same, or of another SC. In the former case, the transaction must be accepted by the network, and it will take time, and a bigger amount of GAS. In the latter case, the execution is immediate, and the cost is lower.

A SC can receive and send Ethers, from and to another SC, or an address. A function belonging to a SC can change its state, can call functions belonging to other SCs, including itself, and can create and send a transaction to an address or to another SC. In the latter case, the transaction is executed immediately. A function which returns a value without changing the state of its SC or sending a transaction, is executed immediately by the EVM and costs nothing. This kind of function is said of type "*view*".

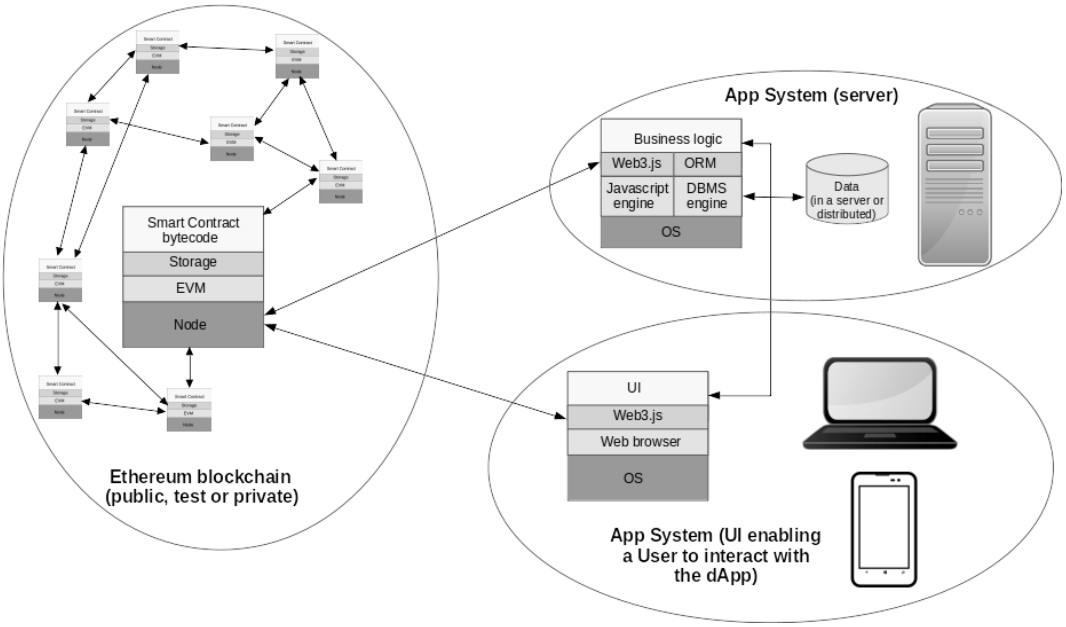

Figure 2: A typical architecture of an Ethereum dApp application. The App System is shown on the right, the blockchain with its SCs on the left.

A typical dApp architecture is shown in Fig. 2. It is composed of a software system running on mobile devices and/or on servers, possibly on the Cloud, exchanging information with users and external devices, which we call "App System". Its User Interface (UI) typically runs on a Web browser. It can have a server component, to store data that cannot be stored in the blockchain, and to perform business computations. In Ethereum, the App System typically communicates with the blockchain using the "web3.js" Javascript library, which manages the creation and dispatch of transactions.

The other component are the SCs running on the blockchain. In a not trivial system, it is composed of various SCs deployed on the blockhain and identified by their Ethereum address.

2.2. Agility and dApp Development

Nowadays, the developments of dApps worldwide share some common characteristics. Several teams involved are typically working on ICO projects, which gathered money through tokens and are about applications of blockchain technology. Other projects are promoted by startups trying to take advantage of the novelty of dApps to develop disruptive solutions, or to get a niche where to thrive. In both cases, they are typically small, self-organizing co-located teams, where experts of system requirements are highly available.

Other characteristics of dApp development is that dApps typically are not life-critical applications, though some of them can be mission-critical. However, the time-to-market and the ability to get an early feedback from the users and the stakeholders are essential, though often the requirements of the dApp initially are only vaguely defined and are subject to change.

All these features make dApp development an ideal candidate for tha use of Agile Methods (AMs). In fact, AMs are suited for small, self-organizing teams, possibly co-located, working on projects whose requirements can change [11]. AMs are considered to be able to deliver quickly and often, as needed by dApp projects.

The most used AM is presently Scrum, which is iterative and incremental, with short iterations (1-4 weeks) [14]. Scrum does not prescribe specific software development practices, but is focused on the process. In short, Scrum, as most other AMs, typically performs requirement elicitation through User Stories (USs), that are short descriptions of how the system answers to inputs from users, or from external devices [22]. USs are mostly gathered at the beginning of the development, but can be modified and augmented at any time. The project advances iteratively implementing a subset of the USs at each iteration. The person in charge of choosing this subset, and explaining their USs to the team is the Product Owner.

Other agile practices that are well suited to dApp development, and that can be used in the proposed process if the team chooses them are:

- *Test Driven Design*: this practice prescribes writing the tests before the code [23], using an automated test suite that can be run whenever needed. For the App System, this is the preferred technique, because it guarantees that the Unit Tests are always present, and their development is not indefinitely postponed if the team is under pressure. For SCs written in Solidity, at the moment the most popular testing environment is Truffle [24].

- *Continuous Integration*: the practice of merging all developer working copies to a shared mainline, even several times a day. Developing dApps, this practice is critical, and it should be practiced both on the App System and the SCs, checking at each merging also how the two systems interact through transactions. This practice requires a development environment provided of a working test blockchain, possibly simulated, to deploy SCs and to test all interactions.

- *Collective code ownership*: this practice allows every developer to intervene on whatever code s/he considers appropriate to modify. With small, dynamic teams as typically happens with dApp development, this practice should clearly be applied. However, often the team members expert in SC development differ from those expert in App System, so their spheres of influence remain separate.

- *Refactoring*: this is the attitude to intervene on the code whenever and wherever it can be improved, improving its design without introducing new features. This practice needs to have an automated test suite, that can be run when the refactoring is made, to assess the absence of unwanted side effects. This is especially needed with the complex architecture of dApps, whose components interacts through transactions.

- *Information Radiators (Cards, Boards, Burndown charts)*: making visible the status of a project using boards that can be observed by everyone and updated in real time, is an practice that is common to all agile projects, and that can obviously greatly benefit also dApp development.

- *Coding Standards*: the practice of strictly following the same coding standard throughout the code, with proper differentiation between App System code and SCs, should be applied to all projects developed following sound SE practices. However, the dynamicity of the teams and the push to quickly develop applications make necessary that the project manager (or the Scrum Master) ensures that this practice is strictly followed.

- *Pair Programming (PP)*: this practice is strictly enforced in "pure" Extreme Programming teams [25]. In our approach, we suggest to use PP in the case the software to be developed is critical, is not yet well understood, or there are new team members to train on the job.

2.3. Security Assessment

In the previous section, we made the case for using agile practices for developing dApps. However, many dApps deal with direct digital currency or token usage, that is with entities that have a direct, real monetary value. In other cases, they may deal with contractual issues, again with strong economic implications, as in the case of document certification, supply chain management, voting systems. Therefore, in most cases dApps are business-critical, and very strict security requirements should be assured. Code inspection, security patterns, and thorough tests must be applied to get a reasonable security level. ABCDE proposed security assessment will be described in detail in Section 4.3.

3. Related Work

SE for dApp development, sometime called Blockchain-Oriented Software Engineering (BOSE) is still in its infancy. The first call for BOSE was made in 2017 by Porru et al. They highligh "*the need for new professional roles, enhanced security and reliability, novel modeling languages, and specialized metrics*", and propose "*new directions for blockchain-oriented software engineering, focusing on collaboration among large teams, testing activities, and specialized tools for the creation of smart contracts*" [10]. They also suggest the adaptation of existing design notations, such as UML, the Unified Modelling Language [26] to unambiguously specify and document dApps.

The book by Xu et al. is perhaps the most complete overview of the engineering aspects of blockchains to date [27]. Among others, it deals with some SE issues, such as the evaluation of the suitability to use a dApp or not, the selection and configuration of the proper blockchain solution (public, permissioned, private), a collection of patterns for the design of blockchain-based applications, and even model-driven generation of SC code. Some of the topics of the book were introduced previously in [28].

Wessling et al. propose a method to find how the architecture of an application could benefit from blockchain technology. They identify the actors involved and how they trust each others to derive a high-level hybrid architecture of a blockchain-based application [29].

Fridgen et al. propose an approach for eliciting use cases in the context of blockchain applications, applying action design research method. Their method is evaluated in four distinct case studies regarding banking, insurance, automotive and construction [30].

Jurgelaitis et al. propose a method based on Model Driven Architecture, which could be used for describing blockchain-based systems using a general language in order to facilitate blockchain development process [31].

A paper by Beller and Hejderup [32] is worth mentioning, though it does not really advocate to use SE practices to develop blockchain applications. Instead, it is about "*how blockchain technology could solve two core SE problems: Continuous Integration (CI) Services such as Travis CI and Package Managers such as apt-get*". The use of SCs to manage agile development, including the automated compensation of developers when their software passes acceptance tests was also proposed by Lenarduzzi et al. [33], [34].

Chakraborty et al. using an online survey got answers from 156 active blockchain software (BCS) developers, finding that "*standard software engineering methods including testing and security best practices need to be adapted with more seriousness to address unique characteristics of blockchain and mitigate potential threat*s" [12]. The same authors published an extended version of the same research, further highlighting that there is a need for "*an array of new or improved tools, such as: customized IDE for BCS development tasks, debuggers for smart-contracts, testing support, easily deployable simulators, and BCS domain specific design notations*" [13]. They found that most BCS developers feel that BCS development is different from traditional one, due to the strict and non-conventional security and reliability requirements, and to other unique characteristics of the dApp development domain (e.g., immutability, difficulty in upgrading the software, operations on a complex, secured, distributed and decentralized network). As anticipated in the Introduction, these findings confirm the expedience to devise a software engineering process such as ABCDE for BCS development.

Regarding dApp security, many publicly available documents, and scientific papers have been already published. Among the most recent ones, the survey of Praitheeshan et al. analyzes the literature about Ethereum SC security, summarizing the main security attacks against SCs, their key vulnerabilities, the security analysis methods and tools [35]. They classify analysis methods in static analysis, dynamic analysis, and formal verification, and discuss the relative pros and cons of these classes, also providing a large bibliography with 160 references. Huang et al. deal with SC security in a broader way, considering also Hyperledger security, and performing a survey from a software lifecycle perspective [36]. After a classification of security issues in SCs, both in Ethereum and Hyperledger Fabric, they consider the securities activities according to the various phases of dApp development (design, implementation, testing before deployment, and runtime monitoring), quoting several references and giving practical advice. These two papers together include references to virtually all the work which have been published about SC security to date.

Various papers have been published to suggest upgrades of Unified Modeling Language [26] notation to enable it to better represent specific application fields. Baumeister et al. described an extension of UML for Hypermedia design, through the addition of a new Navigational Structure Model and new stereotypes [37].

Baresi et al. extend and customize UML with web design concepts borrowed from the Hypermedia Design Model. Hypermedia elements are described through appropriate UML stereotypes [38].

Rocha and Ducasse [39] study SC design and compare three complementary software engineering models – Entity-Relationship diagrams, UML and BPMN. To better represent SC concepts, they propose a simple addition to UML Class Diagrams, that is a small "chain" icon in the UML class representing a contract as a notation to more easily identify it as a blockchain artifact.

4. Proposed Method for dApp Development

4.1. Overall Process

Our approach, ABCDE, takes into account the substantial difference between developing traditional software (the App System) and developing SCs, and separates the two activities. For both developments, ABCDE takes advantage of on an agile approach, because agile methods are suited to develop systems whose requirements are not completely understood since the beginning, or tend to change, as it is the case of dApps. However, a more formal approach with respect to agile development is also added, to address the security and maintenance issues which are very important in dApp development.

The steps of the proposed ABCDE design method, which is currently focused on Ethereum blockchain and Solidity language, are shown in Fig. 3 as UML activity diagram. Note that most steps are in fact performed many times, because the approach is iterative and incremental.

In deeper detail, the proposed development process is the following:

1. **Goal of the system**. Write 10-30 words summing up the goal, and display them in a place that is visible to the whole team. This is a practice that, as far as we know, was introduced by Coad and Yourdon in their 1991 book on object-oriented analysis [40], and that we always found useful. It has some similarities with the "Sprint Goal" that Scrum method prescribes to find and make visible to the team, at the beginning of each iteration [14], but here the goal is found for the whole system.
2. **Find the actors.** Identify the actors who will interact with the dApp system. The actors are human roles, and external systems or devices that exchange information with the dApp to build.
3. **User Stories.** The system requirements are expressed as user stories (USs) [22], to be able to follow the classical agile approach for project management, used in Extreme Programming [25] and Scrum [14]. In this step, the dApp system under development should be considered in full. The decision to develop it using a blockchain, a set of servers, possibly in the cloud, or another architecture, is not important. At this point, we found useful, though not mandatory, to use a UML Use Case Diagram to graphically show the relationships among the actors and the USs. If the decision is taken to implement the system using a blockchain, and related SCs, the following steps are taken.
4. **Divide the system in two subsystems.**
   4.1 The SCs running on the blockchain.
   4.2 The App System, that is the external system that interacts with the blockchain, creating and sending transactions, and monitoring the Events that may happen when a SC executes a function.

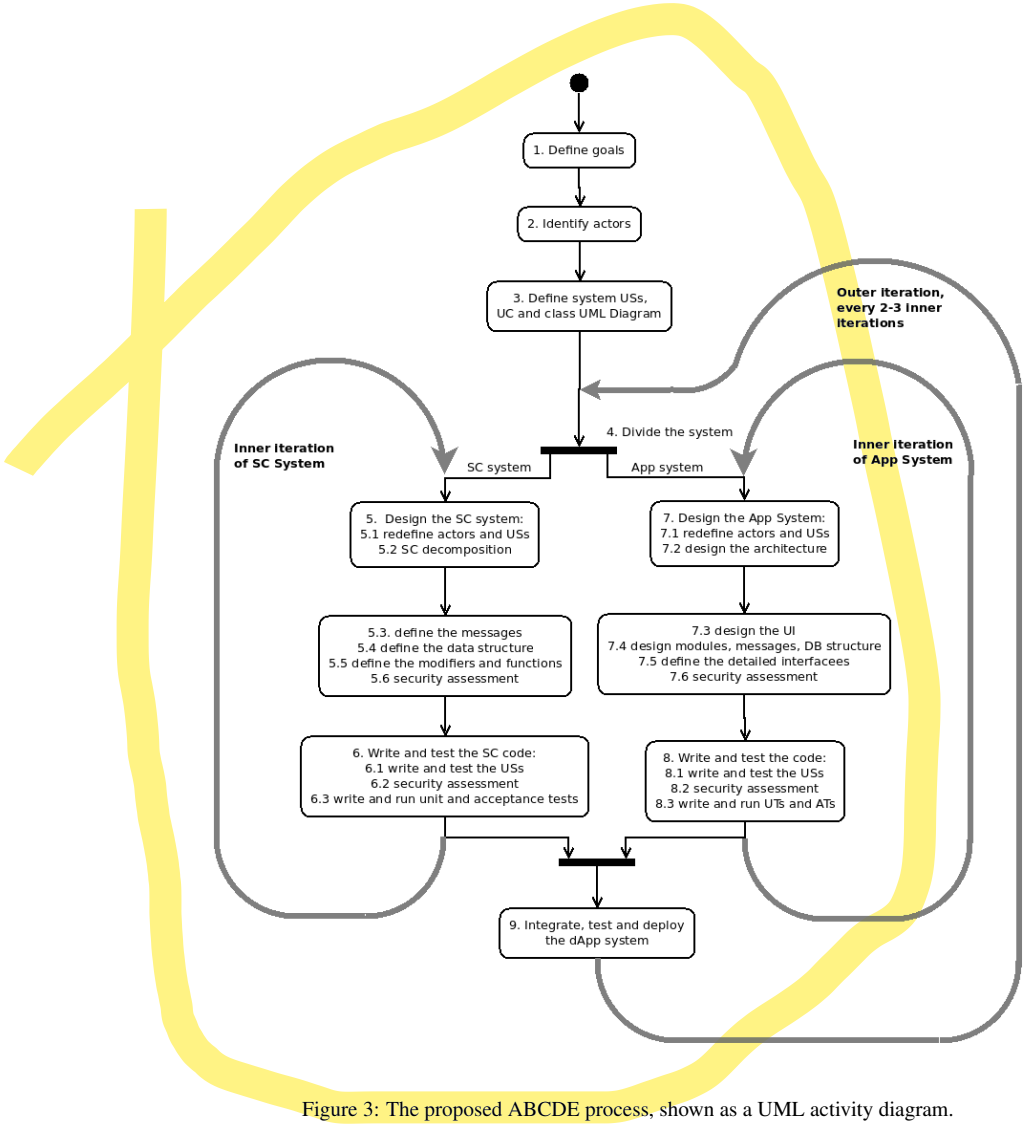

Figure 3: The proposed ABCDE process, shown as a UML activity diagram.

5. **Design of the SCs.**
   This step is about designing the SCs, using in our case the Solidity language. This activity has very peculiar characteristics with respect to standard software design, as highlighted by [12]. The activity is not performed in a single step for the whole SC system, but is performed through iterations that include coding and delivering increments of SCs, which are the USs chosen for each iteration. Its sub-steps are the following:
   - 5.1 Replay steps 2 and 3 (finding Actors and USs) by focusing only on actors directly interacting with the SCs. If external SCs are used by the SCs of the system under development, they should be included among the actors. For each US defined in this step, define also the related acceptance test(s).
   - 5.2 Define broadly the SCs composing the SC subsystem, stating their responsibilities to store information, and the messages they should respond to. For non-trivial systems, you will typically need various interacting SCs. Consider also the use of inheritance for abstracting common features of SCs. Describe in detail the interfaces of the libraries and of the external SCs used. UML class diagrams with proper additions will be used, as shown later in Sec. 4.2.
   - 5.3 Define the flow of messages and Ether transfers among SCs, external SCs and the App System. Use UML sequence diagrams to document these interactions, if they are non-trivial. If needed, define the state changes of SCs using UML statecharts.
   - 5.4 Define in detail the data structure of the SCs, their external interface (Application Binary Interface, ABI) and the relevant events that can be raised by the functions of the SC.

5.5 Define the internal, private functions and the modifiers – special functions that are executed before the functions that apply them, and that usually test the preconditions needed before the function can be safely executed.

5.6 Define the tests and perform the security assessment practices. This is a very important step because, as already explained above, most SCs are very critical and deal with money. Section 4.3 in the followings describes in deeper detail the security assessment we use for Ethereum SCs.

6. **Coding and testing the SC system.** Following the agile approach, the SC system is built and tested incrementally. We advocate following a Scrum approach [14], because it is very effective and popular among developers. In Scrum, a subset of USs are implemented at each iteration. However, also a Lean-Kanban approach is feasible, implementing the USs in a continuous flow, with the work in progress controlled by the Kanban board [41]. The coding and testing activities are:

   6.1 Incrementally write and test the SCs with an agile approach (Scrum or Kanban). Owing to the strict security requirements, typically this activity cannot be performed in a strict incremental way, just implementing one US after another. Instead, starting from the data structure and interfaces of SCs, the overall kernel SC architecture is implemented and tested first. This can be accomplished by using special "User Stories" which are not the description of the interaction with users, but are about the implementation of the architecture of the system. Then, complementary USs can be added.

   6.2 Perform the security assessment of the code written for the increment (see section 5).

   6.3 Write automated Unit Tests (UTs) and Acceptance Tests (ATs) for the SCs and USs implemented, respectively. Add the new tests to the test suite. The most used testing environments for Solidity is Truffle [24]. Run the whole test suite to make sure that the additions did not break the system.

7. **Design of the external interaction subsystem (App System).**
   This step is about designing the App System, which interacts with the users and devices, send messages to the blockchain, and can manage its own repositories (data bases and/or documents). This activity is very similar to designing a standard Web application. It just adds another actor – the blockchain – which can receive (but cannot send) messages and queries. Note that also in this case we must be very careful about security aspects. In fact, often the hacks of dApps systems are made exploiting App System weaknesses, rather that SCs' ones.

   7.1 Redefine the actors and the USs for the App System, starting from those gathered in steps 2 and 3, adding the new actors represented by the SCs that interact with the App System. Define the acceptance tests of the App System.

   7.2 Design the high-level architecture of the App System, including server and client tiers, and detail the way it accesses the blockchain. The access can be done setting up and running one or more nodes of the blockchain, through an external provider, or using a standard wallet.

   7.3 Define the UI of the App System, typically with a responsive approach, so that it can run on both mobile terminals and PCs. Having a fancy UI is of paramount importance to achieve the market success of the whole system. We suggest to perform UI design using well known standard approaches, such as Usage-Centered Design [42] and Interaction Design [43].

   7.4 Define how the App System is decomposed in modules, their interfaces and the flow of messages between them. Define, if needed, the state diagrams of the modules, and the actions they take when events are raised by SCs. Define the structure and memorization of permanent data. Select which data are anchored to the blockchain, by notarization of their hash digest. Define the structure of the data or classes of the App System, including the flow of data and control between modules. The interactions with the SCs must be consistent with the analysis of step 5.3. Since we use and agile approach, this design activity is not performed up-front, but through iterations that include coding and delivering increments of the App System, that is implementation of the USs chosen for the iteration. As the App System is created, the need of additions or updates to the architecture may arise. Due to the above quoted security requirements, this design phase must be quite detailed, and made consistently with the corresponding activities of SCs design. UML class and sequence diagrams can help to design and document this system.

   7.5 Perform a security assessment of the external system, as described below in Sec. 4.3.

8. **Coding and testing the App System.** In parallel to the SCs system, the App System is built and tested. Also in this case, we advocate following a Scrum approach. Alternatively, the team may chose a Lean-Kanban

approach. Of course, the same approach should be used for both SCs and App System development. We stress that, if the developments of SCs and App System are made iteratively, every two or three iterations the results of the two branches must be integrated, as shown in Fig. 3. If a continuous-flow, Lean-Kanban approach is performed, the integration should happen at the completion of every US, in both branches. The activities happening in parallel are:

8.1 Incrementally implement the USs of App System with an agile approach (Scrum or Kanban). This step belongs to the "right flow" of ABCDE (see Fig. 3), and does not differ from the implementation of a Web application using Scrum or Kanban.

8.2 Perform the security assessment of the code written for the increment.

8.3 Write automated Unit Tests (UTs) and Acceptance Tests (ATs) for the USs implemented. Add the new tests to the test suite. Run the whole test suite to make sure that the additions did not break the system.

9. **Integrate, test and deploy the dApp system**. The integration of SCs with App System is performed every 2-3 development iterations of both systems.

4.2. UML diagrams for SCs

Nowadays, the most popular blockchain for dApp development is Ethereum, and the most used language is Solidity. This language is object-oriented (OOPL) because contracts are defined similarly to classes – they have internal variables, and public and private functions. Each SC can inherit from one or more other contracts. With respect to a standard OOPL, Solidity adds specific concepts like events and modifiers, and exhibits limitations in the types available for the SC data structure, and in the management of collections of data – the only collections available so far are the array and the mapping. In the followings, we will describe an adaptation of UML diagrams specific for Solidity 0.5. Possible modifications and extensions for other SC languages will be discussed in the section about future developments.

When designing and documenting SCs, graphic diagrams can be very useful to highlight the connections and the exchange of messages. To this purpose, we advocate the use of a subset of UML diagrams, being UML the universal standard for software design diagrams. However, some specific concepts have to be introduced to account for peculiar SC features. Luckily, UML has an extensibility mechanism called stereotype, which can be used to introduce new concepts, through tagging.

The UML diagrams we considered and modified to model SCs are Class diagrams, and Sequence diagrams. Also, UML Statecharts can be used to graphically represent the various states of a SC, or of a App System module and its transitions. Statecharts, however, do not need any specific stereotype. We already suggested to use also the Use Case diagrams to model actors and related USs (in place of Use Cases).

The Class diagram enables to represent the structure and relationships of SCs. Table 1 shows the stereotypes we introduced in UML class diagrams in order to tag the SC specifities, and their description.

In Solidity, there are no classes, but SCs are very similar to classes – a SC has a data structure composed of variables, and functions able to access these variables. Solidity source code can be used only for creating a SC. This is accomplished by using a special kind of transaction. The other two kinds of transactions are the transfer of Ethers, and the invocation of a function on an existing SC (message). A piece of Solidity code can include several SCs, but a creation transaction can create at most one SC. So, the other SCs can be used to be inherited from the created contract, or to specify the functions that are called on other existing SCs in the same blockchain, accessed through their address. These relationships among SCs can be effectively captured by a UML class diagram.

To address the need to manage complex data, Solidity has the "struct" construct, similar to C, C++ and Java. In UML class diagram, we represent structs as classes, with a proper stereotype and with no bottom operation compartment.

A specific concept of Solidity are *events*, raised when something relevant happens. They can be caught by observer programs, able to act correspondingly. Remember that SCs cannot directly invoke functions of external systems. Another peculiar concept of Solidity are the *modifiers*. These are boolean functions called before a function is executed. They are able to check constraints, and possibly to stop the function execution.

The last four stereotypes of Table 1 are about Solidity collections. Owing to the limitations of blockchain storage, Solidity allows only two kinds of collections – the array and the mapping. These stereotypes denote the kind of collection used for multiple variables of a data structure (middle compartment of UML class symbol), or for implementing

Table 1: Additions to UML class diagram (stereotypes).

Stereotype	**Position**	**Description**
«contract»	Class symbol – upper compartment	Denotes a SC.
«interface»	ditto	A kind of contract holding only function declarations
«library contract»	ditto	A contract taken from a standard library
«enum»	ditto	A list of possible values, assigned to some variable. The values are listed in the middle compartment. The bottom compartment (holding operations) must be empty or absent.
«struct»	ditto	A record, defined in the data structure of a contract and used thereof, able to hold heterogeneous data. The fields are listed in the middle compartment. The bottom compartment must be empty or absent.
«event»	Class symbol, middle compartment	An event that can be raised by the SC, signalling something relevant to external observers.
«modifier»	Class symbol, bottom compartment	A particular kind of guard function, called before another function
«array»	Class symbol, middle compartment, or role of an association	A multiple variable, or 1:n relationship which is implemented using an array.
«mapping»	ditto	The multiple variable, or 1:n relationship is implemented using a generic mapping.
«mapping [*address*]»	ditto	A multiple variable, or 1:n relationship which is implemented using a mapping from an Ethereum address to the value.
«mapping [*uint*]»	ditto	A multiple variable, or 1:n relationship which is implemented using a mapping from a unsigned integer to the value.

an association, aggregation or composition. The array is an ordered set of values, indexed by their position, as in most computer languages. In Solidity, new values can be added to it, but not removed. The corresponding stereotype is "«array»".

The mapping is able to store key-value pairs – the keys being stored as hash values of the actual keys. Given a key, a mapping can efficiently retrieve the value, but it is unable to iterate on its elements, both keys and values. Given the importance of the mapping in Solidity, we introduced three stereotypes to represent a mapping, denoted by the homonymous keyword. The first is the generic mapping; the second is the mapping having an Ethereum address as key, which is very used. The third refers to a common Solidity pattern – using as keys positive, sequential integers, so that it is possible to iterate over them.

The other UML diagram very useful to represent the interactions among SCs and external actors is the Sequence Diagram. These diagrams are used in UML to model messaging. In a blockchain, the relevant messages are related to the transactions, which in turn are sent from external actors, or from SCs to other SCs. Remember that messages are synonyms of "calls of public functions".

A specific characteristic of Ethereum is that messages sent to a SC through a transaction take time (typically 15-20 seconds or more) to be answered. However, if a message is sent to another SC during the execution of a function of a given SC, the time delay is negligible. This happens because the EVM, during the execution of the calling function, is able to locate in the blockchain and call any other SC. To explicitly show this difference, which can be very important for security, GAS consumption and response time, we introduced the stereotypes «trans-msg» and «direct-msg» tagging the message calls sent through a transaction, and directly by a SC, respectively. Note that the fact that UML Sequence Diagrams explicitly represent the flow of time from top to bottom of the diagram can also be

used to quantify the timing difference between the two kinds of messages.

Another peculiarity of Ethereum is that a SC function which does not change the Blockchain is called a "view" function, and can be called immediately and at no cost. Again, this is because the EVM can locate the SC in the blockchain, verify that the function is "view" and call it very quickly and using a negligible amount of resources. All other messages are executed only if proper GAS is paid.

Another kind of message that can be sent is the transfer of Ethers from an address to another. To represent this transfer, we use the Return Message of UML (a dashed arrow), tagged with the stereotype «ethers».

Our Sequence Diagrams represent the message exchange among external actors and SCs, all called *participants*, in a given scenario. The messages among external actors follow the usual UML notation. An external actors, however, can also send Ethers to another. The messages with at least a SC as sender or receiver belong to the following types:

- Transaction: characterized by coming from an external participant to a SC, it is validated and inserted in a block by miners.
- Internal function call: a message sent by a SC to another SC that modifies the blockchain, thus costing GAS; it is represented as the usual "synchronous" or "asynchronous" message of UML Sequence diagrams.
- SC creation: if sent by an external participant it is a transaction, if by another SC it is an internal call; in both cases it implies the call of the constructor of the new SC. In UML notation, creation is represented drawing the message arrow directly into the participant box representing the new SC.
- View function call: a message to a SC which does not modify the blockchain, and costs no GAS; it is denoted by «view» or «pure» stereotype, and can be sent by both kinds of participant.
- Fallback function call: the fallback function is a special function of each SC which is called whenever a function or an Ether transfer fail. This function implements recovery procedures, but can also be used to call whatever function of another contract, through the Proxy pattern. For this reason, we deem important to have a specific «fallback» stereotype.

Note that the above discussed characteristics are specific of Ethereum EVM, irrespectively of the specific high level language used to code the SCs. Table 2 reports the stereotypes we introduced in UML Sequence diagrams to identify the participants sending messages (each having a unique address), and the kinds of messages they exchange.

Table 2: The stereotypes added to UML Sequence diagrams.

Stereotype	**Position**	**Description**
«person»	Participant box	A human role who posts transactions bearing messages, through wallet or some application.
«system»	ditto	An external software system, able to send transactions to the Blockchain.
«device»	ditto	An IoT device, able to send transactions to the Blockchain.
«contract»	ditto	A SC belonging to the system.
«external contract»	ditto	A SC external to the system.
«oracle»	ditto	A particular type of SC, which holds information coming from the external world, provided by a trusted provider.
«account»	ditto	An Ethereum address, just holding Ethers. It can only receive or send Ethers, when its owner activates the transfer.
«trans-msg»	Message	The message is sent using an Ethereum transaction.
«direct-msg»	Message	The message is sent by a SC, so it is executed immediately.
«view» or «pure»	Message	The function called is of type "view" or "pure", so it costs no GAS.
«fallback»	Message	Call to the fallback function. Only called by a SC on itself.
«ethers»	Return Message	The dashed arrow represents a transfer of Ethers, and is can be shown also as a stand-alone message.

4.3. Security assessment for Smart Contracts

Assessing and defining patterns of good programming practice for Smart Contracts for granting security in dApps is still in its infancy and is an ongoing area of research. Nevertheless, based on the programmers' experience and on recent exploited weaknesses –very (in)famous and critical also for the amount of real money involved–, some major advices for security assessment in Smart Contracts have been identified and discussed among the the Solidity developers community. In fact, Ethereum and Blockchain ecosystem are highly new and still somewhat experimental; in addition, SCs are often designed to handle and transfer significant amount of money (in cryptocurrency, but easily exchangeable to real money). Therefore, it is necessary that they correctly achieve their purposes, but it is also crucial that their execution is secure against attacks.

The critical issues regarding the safety of a dApp can be divided in three areas:

- *Issues related to Blockchain itself*: the blockchain itself could be attacked. It is known, for instance, that blockchains using proof-of-work for block generation are subject, at least theoretically, to the so-called "51% attack". Those based on proof-of-stake are vulnerable to other types of attack, for example to "fake stake attack". Using Ethereum technology, the use of the main net lowers the probability of a "51% attack", given the number and the computing power fielded by the miners. Instead, using Ethereum Classic blockchain, a fork derived from Ethereum in 2016, the probability is higher because its miners' computer power is much lower. Using a permissioned blockchain, for instance Ethereum Parity "proof-of-authority", there is no "51% attack", but the blockchain security depends on the honesty and reliability of the validating members, and on their control over their respective IT services. Clearly, this kind of attacks are are more a problem of design choice of the technology to be used than of proper dApp design, so their prevention go beyond the scope of this paper.

- *Issues related to SCs*: the most critical part of a dApp are the SCs, whose bytecode is publicly available, and exposed to all possible exploits. Moreover, developers often lack a full knowledge about implementation and usage of SCs, due to the the fact that this technology is in its early stage, it is evolving fast and is different from traditional development. In literature there are several analyses of possible vulnerabilities related to both Ethereum virtual machine and Solidity language [36] [35] [44]. These are a good starting point for providing a checklist of patterns to verify the SCs under development.

- *Issues related to the App System*: The App System is composed of the server and client side of the dApp, interacting with the SCs on one side, and with human roles and IoT devices and other systems on the other side. It must be designed and implemented with care, but it is somewhat less critical, provided that all best practices related to the security of Web applications are used; a special emphasis must be made to safeguard the access to the private keys of the various actors. We will not cover general Web security practices in this paper.

In the following of this section, we focus on security assurance practices regarding SC design and coding, which are the most critical and less studied among the issues cited above.

4.3.1. General concepts of dApp security

The first and foremost concept in security management is to have a security mindset. The development team(s), and the whole organization, must be fully aware of the importance of security and protection from attacks. Since ABCDE is an agile process, it is based on principles and practices such as: maximize communication, short iterations, refactoring, continuous testing, simplicity, intention-revaling code, use of simple tools. All these practices are also good for security, but Agile means incremental development where USs are continuously completed and tested. This greatly helps productivity, but might be at the expense of security.

A good starting point to focus on security are the Top 10 Proactive Controls of OWASP organization [45]. Those most relevant for dApp security, ordered by importance, are:

- C1: Define Security Requirements. This looks straightforward, but it is not. You must explicitly define the security requirements needed for your system. The requirements can be written as USs, or as non-functional features, and should have acceptance tests in the form of test cases to confirm these requirements have been implemented.

- C2: Leverage Security Frameworks and Libraries. Don't write everything from scratch, but reuse software that is security-hardened, is coming from trusted sources and is maintained up to date.
- C5: Validate All Inputs. This should be performed for user inputs on server-side, because client-side validation can be bypassed. Also, let the SC itself perform validation of key data sent to it through messages.
- C6: Implement Digital Identity. In a dApp environment, digital identities are guaranteed by addresses and by the ownership of the relative private key, so this control is quite straightforward.
- C7: Enforce Access Controls. SC can check access levels of addresses through a mapping, and act accordingly.
- C8: Protect Data Everywhere. In particular, be aware that data stored in a SC are always accessible to read, independently of their visibility.
- C10: Handle All Errors and Exceptions. It is known that even small mistakes in error handling, or forgetting to handle errors can lead to catastrophic failures in distributed systems. This is particularly true for SCs.

Specifically related to SC security are the general guidelines reported in [46], section: "General Philosophy", which complement OWASP ones. Here will just report a short description, and give the names of related security patterns, reported below this list:

1. Prepare for failure. Be able to respond to errors, also in the context of SCs, which cannot be changed once deployed. This is related to patterns 'Emergency stop', 'Rate limit', 'Balance limit' and 'Proxy'
2. Rollout carefully. Try your best to catch and fix the bugs before the SC is fully released. Test contracts thoroughly, and add tests whenever new attack vectors are discovered.
3. Keep SCs simple. Complexity increases the risk of errors, so ensure that SCs and functions are small and modular, reuse SCs that are proven, prefer clarity to performance.
4. Keep up to date. Keep track of new security developments and upgrade to the latest version of any tool or library as quickly as possible.
5. Be aware of blockchain properties. While your previous programming experience is also applicable to SC programming, there are several pitfalls to be aware of.

4.3.2. Security in the design phase

In the design phase, developers must be aware of, and use security patterns, as reported in references [47], [48], [49], which we refer to. Table 6 shows the main security patterns.

Table 3: Main security patterns

ID	Name	Description	Ref.
CEI	Check-effect-interaction	When performing a function in a SC: first, check all the preconditions, then apply the effects to the contract's state, and finally interact with other contracts. Never alter this sequence.	[47]
ES	Emergency stop, also known as "Circuit breaker"	Incorporate an emergency stop functionality into the SC that can be triggered by an authenticated party to disable sensitive functions. This is very useful in the case of major bug or security issue.	[47]
SB	Speed bump	Slow down contract sensitive tasks, so when malicious actions occur, the damage is limited and more time to counteract is available. For instance, limit the amount of money a user can withdraw per day, or impose a delay before withdrawals.	[47]
RL	Rate limit	Regulate how often a task can be executed within a period of time, to limit the number of messages sent to a SC, and thus its computational load.	[47]

MU	Mutex	A mutex is a mechanism to restrict concurrent access to a resource. Utilize it to hinder an external call from re-entering its caller function again.	[47]
BL	Balance limit	Limit the maximum amount of funds held within a SC.	[47]
GC	Guard Check	Ensure that all requirements on a SC state and on function inputs are met. Use properly *assert()*, *require()* and *revert()* to check user inputs, SC state, invariants.	[50]
WF	Withdrawal from Contracts, also known as "Pull over Push"	When you need to send Ethers or tokens to an address, don't send them directly. Instead, authorize the address' owner to withdraw the funds, and let s/he perform the job.	[51], [50]
AU	Authorization	Restrict the execution of code according to the caller address. This is accomplished using mappings of addresses, and is typically checked using modifiers.	[48]
OR	Oracle	An oracle is a SC providing data from outside the blockchain, which are in turn fed to the oracle by a trusted source. Here the security risk lies in how actually the source can be trusted.	[48]
RN	Randomness	Not really a pattern, but some guidelines to simulate randomness in a deterministic environment like that of SCs. It is possible to query an Oracle, to use values not predictable *a priori* as the hash of a block not yet created.	[48]
TC	Time constraint	A time constraint specifies when an action is permitted, depending on the time registered in the block holding the transaction. It is used in Speed bump and Rate limit patterns.	[48]
TE	Termination	Used when the life of a SC has come to an end. This can be done by inserting ad-hoc code in the contract, or calling selfdestruct function. Usually, only the contract owner is authorized to terminate a contract.	[48]
MH	Math	A logic which computes some critical operations, protecting from over-flows, underflows or other undesired characteristics of finite arithmetic.	[48]
PD	Proxy Delegate	Proxy patterns are a set of SCs working together to facilitate upgrading of SCs, despite their intrinsic immutability. A Proxy is used to refer to another SC, whose address can be changed. This approach also ensure that blockchain resources are used sparingly, thus saving GAS.	[49], [50]

Our approach consists in using two security checklists, one to be performed during and after design and design upgrades, the other during coding phases. The aim is to verify that all security patterns and practices concerning known problems are applied. These practices are complementary to the agile practices reported in sections 2.2 and 4.1. Depending on the size of the project and the number of SCs, the checklist can be unique for the system, or you may use a separate checklist for each SC subsystem.

Tables 4 and 5 present the security assurance practices we propose. they describe the checks to be performed, a short description of the vulnerability/vulnerabilities and how to avoid it/them, and one or more references to learn more about the problem. From these tables, it is easy to extract two checklists to be used to perform security assurance during the design and the coding of the SC system, respectively.

Table 4: Security assurance checklist for the design phase

To Check	Description	Ref.	Related patterns
Re-entrancy	Functions that could be called recursively, before the first invocations is finished. This may cause destructive consequences. Ensure state committed before an external call.	[8] [46]	CEI, MU
Dependencies	Use audited and trustworthy dependencies to existing SCs and ensure that newly written code is minimized by using libraries.	[46]	
Multiple Inheritance Caution	Solidity uses the "C3 linearization". This means that when a contract is deployed, the compiler will linearize the inheritance from right to left. Multiple overrides of a function in complex inheritance hierarchies could potentially interact in tricky ways.	[46]	
Include a fail-safe mechanism	It is important to have some way to update the contract in the case some bugs will be discovered. For example, it is possible to have a contract forwarding calls and data to the latest version of the contract.		ES, SB, RL, PD
Limit the amount of ether	If the code, the compiler or the platform has a bug, the funds stored in your smart contract may be lost, so limit the maximum amount. Check that all money transfers are performed through explicit withdrawals made by the beneficiary.		RL, BL, WF
Be careful with randomness	Random number generation in a deterministic system is very difficult. Do not rely on pseudo-randomness for important mechanisms. Current best solutions include hash-commit-reveal schemes (ie. one party generates a number, publishes its hash to "commit" to the value, and then reveals the value later) and RANDAO.	[36] section III-A-7	
Be careful with Timestamp	Be aware that the timestamp of a block can be manipulated by a miner; all direct and indirect uses of timestamp should be analyzed and verified. If the scale of your time-dependent event can vary by 30 seconds and maintain integrity, it is safe to use a timestamp. This include thing like ending of auctions, registration periods, etc. Do not use the *block.number* property as a timestamp.	[35] section IV-C	TC
Never assume that a contract has zero balance	Be aware of coding an invariant that strictly checks the balance of a contract. An attacker can forcibly send ether to any account and this cannot be prevented.	[46]	
Transaction Ordering	Miners have the power to alter the order of transactions arriving in short times. Inconsistent transactions' orders, with respect to the time of invocations, can cause race conditions.	[35] section IV-B	TC

4.3.3. Security in the coding phase

During coding, one major class of problems derives from "external calls", namely from functions which recur to others' SC code for completing their execution. In fact, a SC can call another SC, exploiting the execution of code contained in the latter contract. The pattern can be recursive, so the called SC can in turn perform an external call, and so on. As a consequence, external calls must be treated like calls to 'untrusted' software. They should be avoided or minimized, because some malicious code could be introduced somewhere in a SC belonging to this path, and any external call represents a security risk. A typical risk of such contract interaction is "reentrancy", namely the called contract can call back the calling function before the overall function execution has been completed. This pattern has

been performed in the DAO attack. When it is not possible to avoid external calls, label all the potentially unsafe variables, functions and contracts interfaces as untrusted. Also, follow the "Check-effect-interaction" pattern.

Another important tool for SC security and error handling is the use of *assert()*, *require()* and *revert()* guard functions. They are a very powerful security tool, and are the subject of security pattern "Guard Check" presented in Table 6. In general, use *assert()* to check for invariants, to validate state after making changes, to prevent wrong conditions; if an *assert()* statement fails, something very wrong happened and you need to fix the code. Use *require()* when you want to validate: user inputs, state conditions preceding an execution, or the response of an external call. Use *revert()* to handle the same type of cases as *require()*, but with more complex logic [46].

The most important tool to achieve security and correctness, however, is to apply thorough, automated tests. This is even more crucial when writing SCs, because it is difficult or impossible to update a SC. ABCDE does not prescribe the use ofspecific testing practices, such as Test Driven Design, but highlights the importance of testing. Presently, the most popular testing framework for Ethereum dApps is Truffle, whose website also provides documentation on how to test SCs and App System code – see [24], section: Testing Your Contracts.

The checklist for security assessment in the coding phase is reported in Table 5.

Table 5: Security assurance checklist for the coding phase

To Check	**Description**	**Ref.**	**Related patterns**
External calls	If possible, avoid them. When using low-level call functions (*address.call(), callcode(), delegatecall()* and *send()*) make sure to handle the possibility that the call will fail, by checking the return value. Also, avoid combining multiple ether transfers in a single transaction. Mark untrusted interactions: name the variables, methods, and contract interfaces of the functions that call external contracts, in a way that makes it clear that interacting with them is potentially unsafe.	[46]	CEI, MU, GC, WF
Prevent overflow and underflow	If a balance reaches the maximum uint value it will circle back to zero; similarly, if a uint is made to be less than zero, it will cause an underflow and get set to its maximum value. One simply solution is to use a library like *SafeMath.sol* by OpenZeppelin.	[35] section III-C	MH, GC
Beware of rounding errors	All integer divisions round down to the nearest integer. Check that truncation does not produce unexpected behaviour (locked funds, incorrect results).	[46]	MH, GC
Validate inputs to external and public functions	Make sure the requirements are verified and check for arguments.	[35] section IV-F	GC
Prevent unbounded loops	The gas consumed increases with each iteration until it hits the block's gasLimit, stopping the execution.	[50]	
tx.origin	It is a global variable that returns the address of the message sender. Do not use *tx.origin* as an authorization mechanism.	[52] section 3.1	
Fallback functions	Fallback functions are called when a contract receive a message without arguments and when no other function matches. You should keep them simple and check that the data is empty to avoid malicious invocation.	[46] [35] section IV-A	CEI, MU, GC
Check if built-in variables or functions were overridden	It is currently possible to override built-in globals in Solidity, such as . This allows SCs to override the functionality of built-ins such as *msg* and *revert()*. Although this is intended, it can mislead users of a SC, so the whole SC code must be checked.	[46]	GC

Use interface type instead of the address for type safety	When a function takes a contract address as an argument, it is better to pass an interface or contract type rather than raw *address*. If the function is called elsewhere within the source code, the compiler it will provide additional type safety guarantees.	[46]	GC
Enforce invariants with assert()	An assert guard triggers when an assertion fails - for instance an invariant property changing. You can verify it with a call to *assert()*. Assert guards should be combined with other techniques, such as pausing the contract and allowing upgrades. (Otherwise, you may end up stuck, with an assertion that is always failing.)	[46]	GC
Lock pragmas to specific compiler version	Contracts should be deployed with the same compiler version and flags that they have been tested with, so locking the version helps avoid the risk of undiscovered bugs.	[46]	
Fix compiler warnings	Take warnings seriously and fix them. Always use the latest version of the compiler to be notified about all recently introduced warnings.	[51]	
Testing	Be sure to have a 100% text coverage and cover all critical edge cases with unit tests. Do not deploy recently written code, especially if it was written under tight deadline.		

4.4. Gas optimization

Besides security, another important factor of SCs that must be carefully designed since the beginning is their cost. Creating SCs and writing permanent data in a public blockchain can be very costly, so it is important to keep them to a minimum, and to limit the transactions that write or modify these data. Also, the messages exchanged among the App System and the SCs, and among SCs, must be properly designed and well documented. Table 6 shows some specific patterns that can be used to save GAS.

Note that in Ethereum the maximum size of the bytecode of a SC is restricted to 24 KBytes by the standard EIP 170 (see section 13.4.2 of [27]). For serious SCs, that size limit can be hit easily, so many of the GAS saving patterns are useful also to make a SC viable.

Table 6: Main GAS saving patterns

ID	Name	Description	Ref.
PD	Proxy Delegate	When you need to call external SCs, do not include their code. Include their interface and use the Proxy pattern, which uses the fallback function to call the SC functions. This is the same pattern also shown in Table 6	[49]
LS	Limit Storage	Limit data stored in the blockchain. Store non-permanent data in memory. Avoid changing storage data during computations – change them only after all the calculations.	[53]
PK	Pack your variables	In Ethereum, you pay GAS for every storage slot of 256 bits you use. You can pack as many variables as you want in it, but you must order their declaration properly. Use integers smaller than 256 bits only if you have many to pack. If not, using 256 bits integers avoids the needed conversion to 256 bits, which costs GAS. Remember that elements in memory and call data are not packed. Use datatype *bytes32* rather than *bytes* or *string*, if possible. Limit constant strings, for instance those used in *require()* to explicit the error, to fit in 32 bytes.	[53]

DV	Delete variables no more needed	If you donâĂŹt need a variable anymore, delete it using the *delete* keyword. In Ethereum, you get a GAS refund for freeing up storage space.	[53]
NI	Do not initialize variables with default values	All variables are initialized to zeroes at no cost. Do not explicitly initialize them to zero, or a value is given to them anyway when they are used.	[53]
MP	Use Mappings	To manage lists of data, use mappings with integer key and not arrays. This is known to save blockchain space.	[53]
EP	Execution Paths	Thoroughly examine all possible execution paths, looking for code whose execution can be spared. Avoid repetitive checks of variables. Logical operators '||' and '&&' evaluate only the first operand if the second is not needed, so order them to maximize the probability that only one operand is computed.	[53]
LE	Limit external calls	Limit calls to other SCs. Note that calling *external* functions is cheaper than calling *public* functions. The cheapest calls, however, are those to *internal* functions.	[53]
LM	Limit modifiers	. The code of modifiers is "inlined" inside the modified function, thus costing GAS. Internal functions, on the other hand, are not inlined but called as separate functions. They are very slightly more expensive in run time but save a lot of redundant bytecode in deployment, if used more than once.	[53]
UL	Use libraries	. The bytecode of external libraries is not made part of your SC, thus saving GAS. However, calling them is costly and has security issues. Use libraries for complex tasks.	[53]
EL	Event Log	If the App System needs to retrieve information about past events, that is not useful for SC execution, let the app directly access the Event Log in the blockchain. Note that if the event happened far in time, the time to retrieve it may be long.	[53]

5. Experimental Validation

The development process which later was named ABCDE was first devised in 2018 [54], and since then it has been used in several project carried on in our University group, and in firms we are consulting. Among the projects which were developed, or which are in development, we may quote a system to trace the provenance of foods, a supply chain management system, a system to manage temporary job contracts, a voting system which was used to reward the best presentation at a conference, another one managing voting in firm shareholders' and board of directors meetings, a system to manage energy exchange in local networks of electricity producers and consumers, a system to automate agile software development [34].

The feedback of dApp developers using ABCDE method was generally positive, and was used to improve the method – especially concerning security and GAS optimization practices.

Here we present, as an example of ABCDE usage, a simplified version of a dApp application aiming to implement a decentralized exchange (DEX) for tokens managed on Ethereum blockchain. A DEX is a system enabling the exchange of different tokens between two holders, who interact directly, without intermediaries. We started from the well-known 0x protocol project, the subject of a successful ICO held in 2017. The specification of the DEX can be found in the 0x Whitepaper [55]. We present a simplified version of the whole system. In particular, we dropped the part related to the protocol token (Section 4 of the Whitepaper). Moreover, for the sake of brevity, we will not present the coding phases (phases 6, 8 and 9), but we stop at the end the design phases (phases 5 and 7). The steps of ABCDE are presented below.

1. **Goal of the system**. *To manage a decentralized exchange, able to enable pairs of ERC20 and ERC721 token holders to exchange their tokens at an agreed rate on the Ethereum blockchain*.
2. **Actors**. The system has the following actors:
   - **Trader**: owner of tokens, wishing to post an offer, or to accept a posted offer.
   - **Maker**: a trader who posts an offer to sell a given amount of her/his tokens, in exchange to tokens of another type, at a given exchange rate.
   - **Taker**: a trader who accepts the offer of a Maker.
   - **Relayer**: a system which facilitates signaling between market participants by hosting and propagating an order book of the offers.
   - **DEX**: smart contract(s) on the Ethereum blockchain which accept orders signed by both a Maker and Taker, and activate the exchange of tokens.
   - **Token**: a SC on the Ethereum blockchain, managing a given token according to the ERC20 or ERC721 protocols.

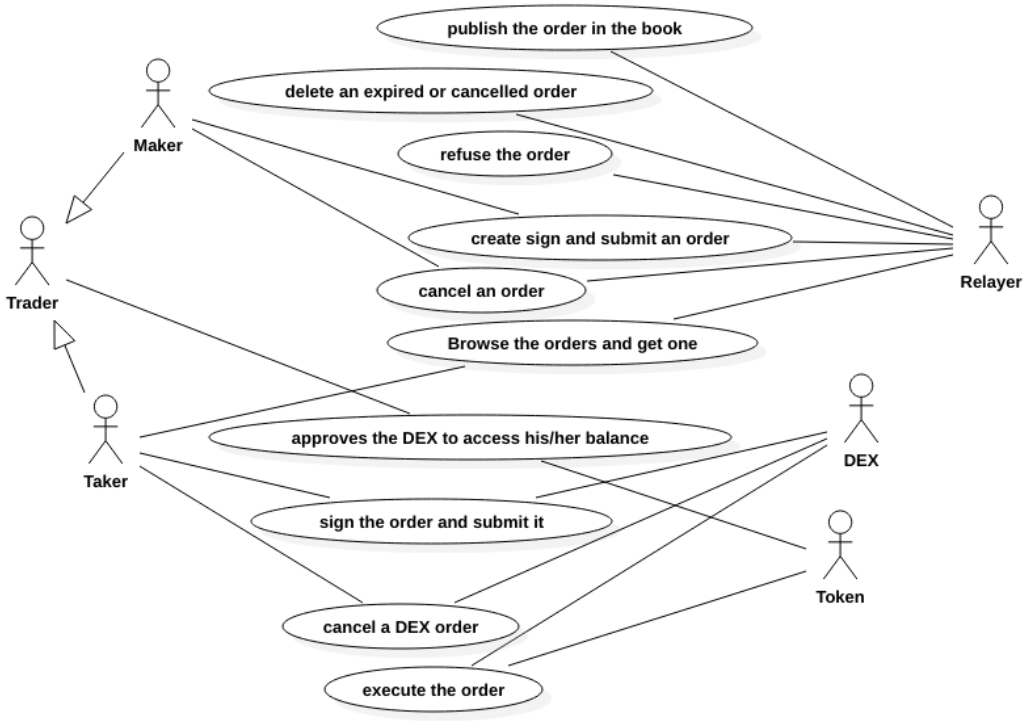

Figure 4: The User Stories of the DEX system specification.

3. **User Stories**. Fig. 4 shows the actors and the USs they are involved in, using a UML Use Case diagram, where the use cases are in fact USs. Note that these USs just specify the DEX, and do not depend on the specific technology used to implement it, except for the Ethereum blockchain, which the DEX necessarily has to interact with. Here we have no room to show the USs in detail, so we refer the readers to the above quoted Whitepaper [55]. Instead, in Fig. 5 we show the UML class diagram derived by an analysis of the given USs. This diagram is not bound to a specific implementation of the relayer system, but just shows schematically the entities, the data structures and the operations emerging from the USs of Fig. 4.
4. **Divide the system into SC and App subsystems**. In this case the subdivision is trivial, because the Relayer system is a typical Web application, whereas the DEX and the Tokens are Smart Contracts by design. The

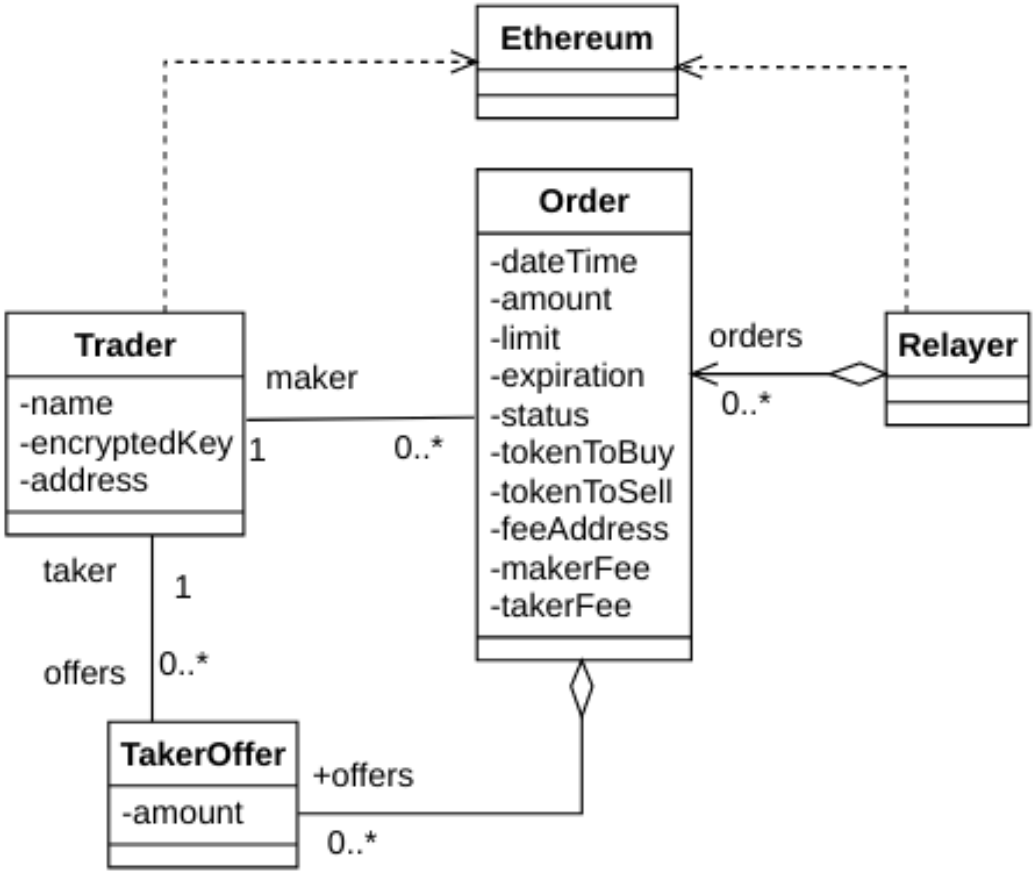

Figure 5: The standard UML class diagram derived from the USs.

USs of the external app subsystem are the same of those reported in Fig. 4, except the last one ("execute the order"), which is carried on solely by SCs. As regards the blockchain subsystem, the US to implement are basically the messages to submit an order, or to cancel an order, sent by a Taker to the DEX. In practice, the actual implementation of the DEX contracts made by 0X Team is very complex, due to the strict security requirements, and to the many checks that must be made before performing the actual token transfer.

5. **Design of the SC subsystem**. The SC system is very complex, and a detailed description of its architecture is well beyond the scope of this paper. We report in Fig. 6 just a simplified UML class diagrams showing some of the actual SCs, to show some of the specific stereotypes used to document an SC system, as described in section 4.2. The modifiers and the events enforcing the constraints relevant for the DEX are shown in Table 7 and Table 8, respectively.
   Fig. 7 shows a UML sequence diagram representing the interactions among most Actors of the systems, when a Taker accepts an order seen in the Relayer's book, and sends it to the DEX for execution, including the messages exchanged among the SCs.
6. *Omitted*.
7. **Design of App System**. The App System is composed of the software able to present the present offers of tokens posted by the takers, and of the software used by takers and makers, respectively to post, modify or delete offers, and to accept offers. The latter software must be provided of a wallet able to store Ethers and send transactions to Ethereum blockchain. The design of this subsystem includes that of its user interfaces. The system is fairly complex, and the wallets must be designed and implemented using strong security practices. We will not dig further into this subsystem because, except for the wallet, it is a standard, Web-based system.

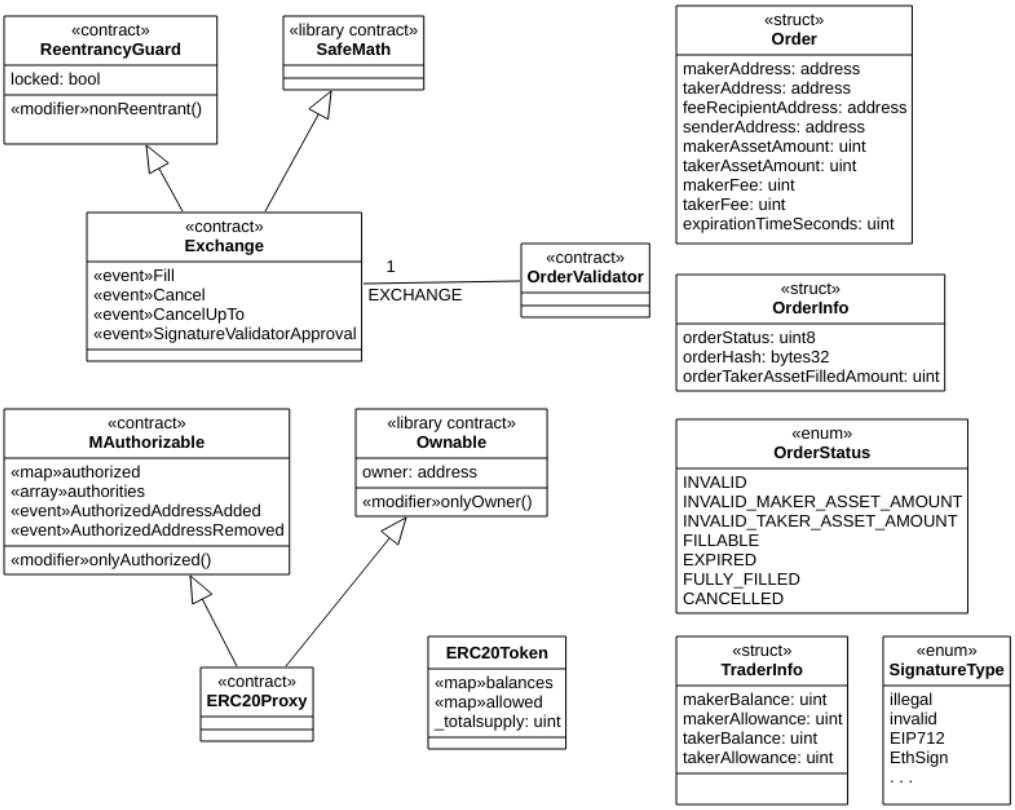

Figure 6: The modified UML diagram, showing the structure of the required SCs of the DEX system.

8. *Omitted.*
9. *Omitted.*

6. Conclusion and Future Work

Despite the huge effort presently ongoing in developing dApps, software engineering practices are still poorly applied in software development of blockchain systems. The field is in fact still in its infancy, and tools or techniques for modeling and managing the peculiarities a software developer must face when dealing with blockchain-oriented software systems are still matter for researchers. Tools and techniques of traditional software engineering have not yet been adapted and modified to adhere to this new software paradigm. A sound software engineering approach might greatly help in overcoming many of the issues plaguing blockchain development providing developers with instruments similar to those typically used in traditional software engineering to afford architectural design, security issues, testing planes and strategies and to improve software quality and maintenance.

Researchers in software engineering have a big opportunity to start studying a field that is very important and brand-new exploiting concepts, tools, instruments and ideas already consolidated in software engineering and changing and adapting them to this new software technology.

This work, whose a first version was presented in [54], moves toward this direction providing a full modeling of interactions among traditional software and blockchain environment, including Class diagrams, Statecharts, USâĂŹs diagrams, Sequence diagrams, Smart Contracts diagrams – all specialized for blockchain application development. It also provides a general scheme for managing blockchain development processes, and a simplified example of a Distributed Exchange Smart Contract, taken from a real set of SCs implementing a DEX.

Table 7: The *modifiers* of the SCs.

Modifier	Action – Notes
onlyOwner()	Enforces that the sender of the message is the owner of the contract. Inherited by Ownable standard contract
nonReentrant()	Enforces that the message is not recursively sent
onlyAuthorized()	Enforces that the sender of the message belongs to a list of authorized addresses managed by the same SC

Table 8: The *events* of the SCs.

Event	Action – Notes
Fill	An order has been filled in the DEX.
Cancel	An order has been cancelled in the DEX.
CancelUpTo	An order has been partially cancelled from the DEX.
AuthorizedAddressAdded	An address has been added to the list of authorized ones.
AuthorizedAddressRemoved	An address has been removed from the list of authorized ones.

We believe that our work can be really valuable to blockchain firms, including ICO startups, that could develop a competitive advantage using SE (BOSE) practices since the beginning. The proposed method has also the potential to be applied to other SC environments, such as Hyperledger, Ripple and others, and we are exploring these extensions.

7. ACKNOWLEDGMENTS

This work was partially founded by the AIND project (Native Digital Administrations and Enterprises), funded by Sardinia Region, PIA call 2013, E.U. P.O. FESR 2007/2013, n. 3706 Rep. n. 316, 22/04/2016.

References

[1] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, 2008. Url="https://bitcoin.org/bitcoin.pdf", last accessed: August 15, 2019.
[2] G. Wood, Ethereum: A secure decentralised generalised transaction ledger, 2014. Url="https://ethereum.github.io/yellowpaper/paper.pdf", last accessed (updated version): August 15, 2019.
[3] N. Szabo, Smart contracts: Formalizing and securing relationships on public networks, First Monday 2 (1997). Url="https://ojphi.org/ojs/index.php/fm/article/view/548".
[4] M. Swan, Blockchain: Blueprint for a new economy, O'Reilly Media, Inc., 2015.
[5] M. Biella, V. Zinetti, Blockchain technology and applications from a financial perspective, Unicredit Tehcnical Report (2016).
[6] G. Fenu, L. Marchesi, M. Marchesi, R. Tonelli, The ico phenomenon and its relationships with ethereum smart contract environment, in: Blockchain Oriented Software Engineering (IWBOSE), 2018 International Workshop on, IEEE, pp. 26–32.
[7] U. Chohan, The Problems of Cryptocurrency Thefts and Exchange Shutdowns, Technical Report, Discussion Paper Series: Notes on the 21 st Century, School of Business and Economics, University of New South Wales, Canberra, 2018.
[8] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on ethereum smart contracts (sok), in: M. Maffei, M. Ryan (Eds.), Principles of Security and Trust, Springer Berlin Heidelberg, 2017, pp. 164–186.
[9] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, R. Hierons, Smart contracts vulnerabilities: A call for blockchain software engineering?, in: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE).
[10] S. Porru, A. Pinna, M. Marchesi, R. Tonelli, Blockchain-oriented software engineering: challenges and new directions, in: Proceedings of the 39th International Conference on Software Engineering Companion, IEEE Press, pp. 169–171.
[11] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al., Manifesto for agile software development (2001).
[12] P. Chakraborty, R. Shahriyar, A. Iqbal, A. Bosu, Understanding the software development practices of blockchain projects: A survey, in: ESEM 2018, October 11âĂŞ12, 2018, Oulu, Finland, ACM.
[13] A. Bosu, A. Iqbal, R. Shahriyar, P. Chakraborty, Understanding the motivations, challenges and needs of blockchain software developers: a survey, Empirical Software Engineering 24 (2019) 2636–2673.
[14] K. Schwaber, M. Beedle, Agile Software Development with Scrum, Pearson, 2001.
[15] K. Beck, Test Driven Development: By Example, Addison-Wesley Professional, 2002.
[16] M. Fowler, Refactoring: Improving the Design of Existing Code (2nd Edition), Addison-Wesley Professional, 2018.
[17] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, H. Wang, Blockchain challenges and opportunities: a survey, International Journal of Web and Grid Services 14 (2018) 352–375.

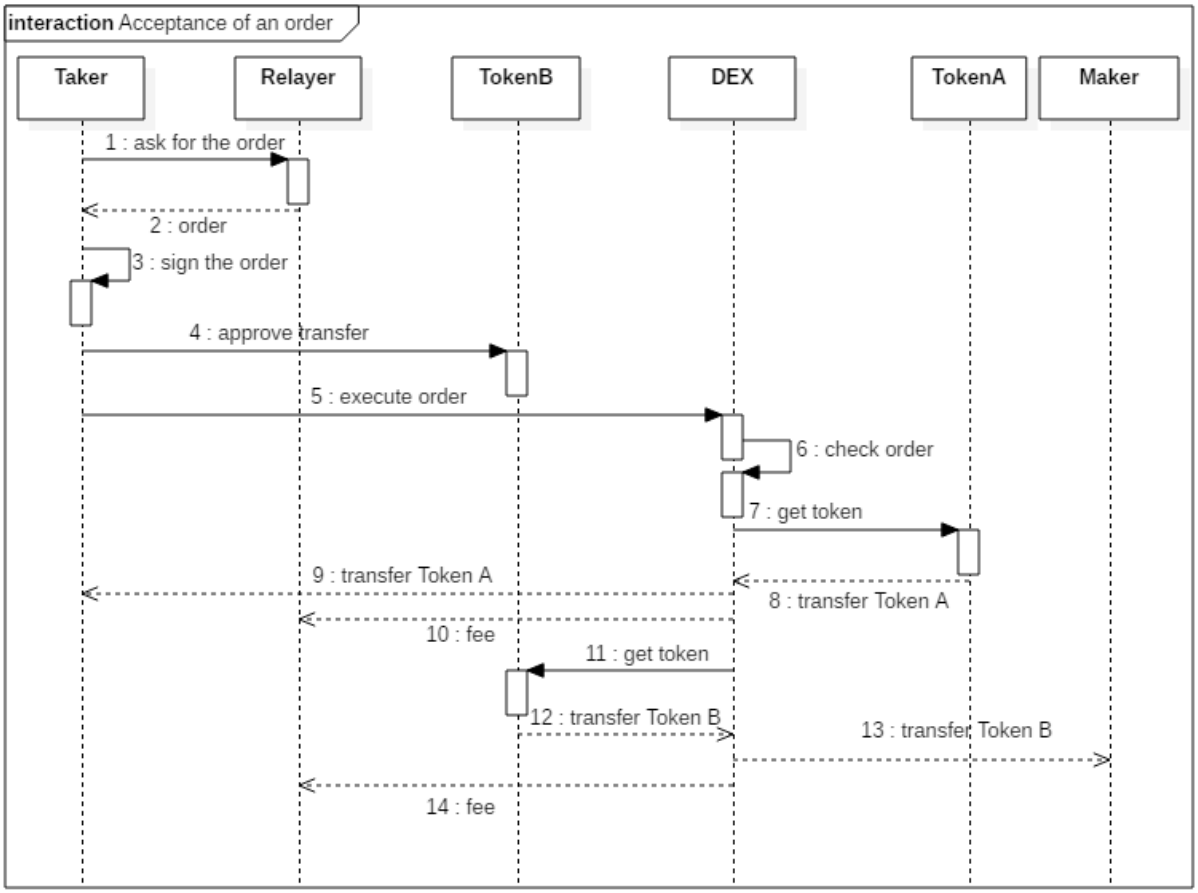

Figure 7: The UML sequence diagram showing a Taker accepting an offer and sending it to the DEX for execution.

[18] S. Tikhomirov, Ethereum: state of knowledge and research perspectives, in: International Symposium on Foundations and Practice of Security, Springer, pp. 206–221.

[19] Coinmarketcap website, 2019. Url="https://coinmarketcap.com/tokens/", last accessed: August 12, 2019.

[20] State of the dapps website, 2019. Url="https://www.stateofthedapps.com/stats", last accessed: November 16, 2019.

[21] C. Dannen, Introducing Ethereum and Solidity, Springer, 2017.

[22] M. Cohn, User Stories Applied: For Agile Software Development, Addison-Wesley Professional, 2004.

[23] D. Janzen, H. Saiedian, Test-driven development concepts, taxonomy, and future direction, Computer 38 (2005) 43–50.

[24] Truffle website, 2019. Url="https://www.trufflesuite.com/", last accessed: October 6, 2019.

[25] K. Beck, Extreme programming explained: embrace change, Addison-Wesley professional, 2000.

[26] J. Rumbaugh, G. Booch, I. Jacobson, The unified modeling language reference manual, Addison Wesley, 2017.

[27] X. Xu, I. Weber, M. Staples, Architecture for Blockchain Applications, Springer, 2019.

[28] X. Xu, I. Weber, M. Staples, L. Zhu, J. Bosch, L. Bass, C. Pautasso, P. Rimba, A taxonomy of blockchain-based systems for architecture design, in: Software Architecture (ICSA), 2017 IEEE International Conference on, IEEE, pp. 243–252.

[29] F. Wessling, C. Ehmke, M. Hesenius, V. Gruhn, How much blockchain do you need? towards a concept for building hybrid dapp architectures, in: WETSEB 2018-1st International Workshop on Emerging Trends in Software Engineering for Blockchain.

[30] G. Fridgen, J. Lockl, S. Radszuwill, A. Rieger, A. Schweizer, N. Urbach, A solution in search of a problem: A method for the development of blockchain use cases, in: 24th Americas Conference on Information Systems (AMCIS), New Orleans, USA, August 2018.

[31] M. Jurgelaitis, V. Drungilas, L. Ceponiene, R. Butkiene, E. Vaiciukynas, Modelling principles for blockchain-based implementation of business or scientific processes, in: Proceedings of the International Conference on Information Technologies, IVUS 2019, pp. 43–47.

[32] M. Beller, J. Hejderup, Blockchain-based software engineering, in: Proceedings of the 41th International Conference on Software Engineering Companion, IEEE Press, pp. 53–56.

[33] V. Lenarduzzi, I. Lonesu, M. Marchesi, R. Tonelli, Blockchain applications for agile methodologies, in: Proceedings of the 19th International Conference on Agile Software Development: Companion, XP 2018, ACM, New York, NY, USA, 2018, pp. 30:1–30:3.

[34] M. Marchesi, G. Destefanis, V. Lenarduzzi, M. Lunesu, M. Ortu, A. Pinna, R. Tonelli, Agile software development automated by blockchain smart contracts, in: Proceedings of the Software Engineering Conference Russia, SECR 2019, ACM, New York, NY, USA, 2019.

[35] P. Praitheeshan, L. Pan, J. Yu, J. Liu, R. Doss, Security analysis methods on ethereum smart contract vulnerabilities: A survey, arXiv preprint arXiv:1908.08605 (2019).
[36] Y. Huang, Y. Bian, R. Li, L. Zhao, P. Shi, Smart contract security: A software lifecycle perspective, IEEE Access, 7 (2019).
[37] H. Baumeister, N. Koch, L. Mandel, Towards a uml extension for hypermedia design, in: International Conference on the Unified Modeling Language, Springer, pp. 614–629.
[38] L. Baresi, F. Garzotto, P. Paolini, Extending uml for modeling web applications, in: System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on, IEEE, pp. 10–pp.
[39] H. Rocha, S. Ducasse, Preliminary steps towards modeling blockchain oriented software, in: WETSEB 2018-1st International Workshop on Emerging Trends in Software Engineering for Blockchain.
[40] P. Coad, E. Yourdon, P. Coad, Object-oriented analysis, volume 2, Yourdon press Englewood Cliffs, NJ, 1991.
[41] D. J. Anderson, Kanban: successful evolutionary change for your technology business, Blue Hole Press, 2010.
[42] L. L. Constantine, L. A. Lockwood, Software for use: a practical guide to the models and methods of usage-centered design, Pearson Education, 1999.
[43] H. Sharp, Y. Rogers, J. Preece, Interaction design: beyond human-computer interaction, 5th edition, John Wiley & Sons, 2019.
[44] J. Liu, Z. Liu, A survey on security verification of blockchain smart contracts, IEEE Access, 7 (2019).
[45] K. Anton, J. Manico, J. Bird, OWASP Proactive Controls for Developers, Technical Report, Open Web Application Security Project (OWASP), 2018.
[46] Consensys solidity best practices website, 2019. Url="https://consensys.github.io/smart-contract-best-practices/", last accessed: November, 2019.
[47] M. Wohrer, U. Zdun, Smart contracts: Security patterns in the ethereum ecosystem and solidity, in: Blockchain Oriented Software Engineering (IWBOSE), 2018 International Workshop on, IEEE, pp. 2–8.
[48] M. Bartoletti, L. Pompianu, An empirical analysis of smart contracts: platforms, applications, and design patterns, in: B. M. et al. (Ed.), Financial Cryptography and Data Security. FC 2017. Lecture Notes in Computer Science, Springer, Cham, 2017, pp. 494–509.
[49] Proxy patterns, 2019. Url="https://blog.openzeppelin.com/proxy-patterns/", last accessed: November, 2019.
[50] Ethereum smart contract security best practices website, 2019. Url="https://ethereum-contract-security-techniques-and-tips.readthedocs.io/en/latest/", last accessed: November, 2019.
[51] Solidity website, 2019. Url="https://solidity.readthedocs.io/en/v0.5.13/index.html", last accessed: November, 2019.
[52] A. Mense, M. Flatscher, Security vulnerabilities in ethereum smart contracts, in: Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services, iiWAS2018, ACM, New York, NY, USA, 2018, pp. 375–380.
[53] M. Gupta, Solidity gas optimization tips, 2018. Url="https://mudit.blog/solidity-gas-optimization-tips/", last accessed: November, 2019.
[54] M. Marchesi, M. Marchesi, R. Tonelli, An agile software engineering method to design blockchain applications, in: Proceedings of the Software Engineering Conference Russia, SECR 2018, ACM, New York, NY, USA, 2018.
[55] W. Warren, A. Bandeali, 0x An open protocol for decentralized exchange on the ethereum blockchain, 2017. Url="https://0xproject.com/pdfs/0x_white_paper.pdf", last accessed: November, 2019.