Blockchain: Research and Applications 1 (2020) 100002

Contents lists available at ScienceDirect

Blockchain: Research and Applications

journal homepage: www.journals.elsevier.com/blockchain-research-and-applications

ABCDE –agile block chain DApp engineering

Lodovica Marchesi, Michele Marchesi, Roberto Tonelli *

DMI, University of Cagliari, Italy

ARTICLE INFO

Keywords:
Blockchain
Smart contracts
Blockchain-oriented software engineering
UML
DApp design

ABSTRACT

Blockchain software development is becoming more and more important for any modern software developer and IT startup. Nonetheless, blockchain software production still lacks of a disciplined, organized and mature development process, as demonstrated by the many and (in)famous failures and frauds occurred in recent years. In this paper we present ABCDE, a complete method addressing blockchain software development. The method considers the software integration among the blockchain components – smart contracts, libraries, data structures – and the out-of-chain components, such as web or mobile applications, which all together constitute a complete DApp system. We advocate for ABCDE the use of agile practices, because these are suited to develop systems whose requirements are not completely understood since the beginning, or tend to change, as it is the case of most blockchain-based applications. ABCDE is based on Scrum, and is therefore iterative and incremental. From Scrum, we kept the requirement gathering with user stories, the iterative-incremental approach, the key roles, and the meetings. The main difference with Scrum is the separation of development activities in two flows – one for smart contracts and the other for out-of-chain software interacting with the blockchain – each performed iteratively, with integration activities every 2–3 iterations. ABCDE makes explicit the activities that must be performed to design, develop, test and integrate smart contracts and out-of-chain software, and documents the smart contracts using formal diagrams to help development, security assessment, and maintenance. A diagram derived from UML class diagram helps to effectively model the data structure of smart contracts, whereas the exchange of messages between the entities of the system is modeled using a modified UML sequence diagram. The proposed method has also specific activities for security assessment and gas optimization, through systematic use of patterns and checklists. ABCDE focuses on Ethereum blockchain and its Solidity language, but preserves generality and with proper modifications might be applied to any blockchain software project. ABCDE method is described in detail, and an example is given to show how to concretely implement the various development steps.

1. Introduction

The so-called “decentralized applications”, or “DApps”, is a trending area 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 the Ethereum blockchain, a network whose nodes are also able to run Turing-complete programs [2] called “smart contracts” (SCs) following an idea of Nick Szabo [3]. SCs are general computer programs, though with some specific features. The original 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.

This interest led to a substantial amount of money flooding into Blockchain ventures, including 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

* Corresponding author.
E-mail addresses: lodovica.marchesi@unica.it, lodovica.marchesi@unica.it (L. Marchesi), marchesi@unica.it, marchesi.michele@gmail.com (M. Marchesi), roberto.tonelli@dsf.unica.it (R. Tonelli).

Production and Hosting by Elsevier on behalf of KeAi

https://doi.org/10.1016/j.bcra.2020.100002
Received 2 September 2020; Received in revised form 6 November 2020; Accepted 26 November 2020

invitation.

Note that a blockchain is a kind of DL technology (DLT), but not all DLT is based on a blockchain, because a DL can use other cryptographic approaches to hold immutable the transactions history (it does not need a consensus mechanism to pack transactions history into a chain of blocks). In the following, we will mainly talk of "blockchain", but this often will be interchangeable with "DLT".

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 DLs – are based on developing software inside a new paradigm. In this context, we assisted to a run to be the first on the market, as always happens with new technology waves. This led to quick application development, often neglecting good development practices, and even comprehensive testing and security assessment.

Some 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 [4], and also smart contracts were often exploited, taking advantage of their novelty and of the hurried software development [5,6].

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 latter, in the context of blockchain development, we stress the importance of requirement elicitation, system design, specific notations, testing and security assessment. In essence, we need blockchain-oriented software engineering (BOSE) [7].

In this paper, we try to cope with these issues proposing a software development process to gather the requirements, to analyze, design, develop, test and deploy DL applications. In particular, we present a development process for applications based on smart contracts running on a blockchain, that is for "DApps". We designed this process specially referring to Ethereum blockchain, and to its Solidity programming language, because it is by far the most popular DApp platform (see Section 2). 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", for Agile Block Chain Dapp Engineering. ABCDE is an agile software development process, meaning that it follows the principles of Agile Manifesto [8]. However, we complement the agile process with a more formal approach, using UML diagrams with a specific notation for smart contracts, and a specific checklist for security assessment.

The first question we have to answer regarding ABCDE is "why a new process"? Why not to use an existing process, waterfall, iterative or agile, for DApp development? The answer to this question stems from the observation that smart contracts and the software based and running on a blockchain are application-specific software. SCs run on all the nodes hosting a blockchain, and their execution has the strong constraint that all outputs and state changes resulting from SC execution must be the same in all nodes.[1] 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 direct 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. Among these, the concepts of "address", signing with the private key behind the address, "gas" consumption, SC immutability once deployed, cryptocurrency transfer and "oracle".

Moreover, referring to Solidity, 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.

A further confirmation about the need for specific SE in Blockchain software development comes from two papers by Chakraborty et al. [9,10], which 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.

These specificities led us to conclude that a new methodology of software development is needed for DApp development. In fact, ABCDE is not entirely new, but it is a significant extension of classical agile methods, such as Scrum [11]. 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 smart contracts, and checklists for security assessment.

A first version of the proposed development process, not yet named ABCDE was proposed at a conference in 2018 [12]. This paper greatly extends and updates that previous paper. In particular, we introduced more additions to UML diagrams, we better specified security checklists and introduced gas saving checklists, and revised the whole method based on the feedback received by developers who used it.

The main contribution of our work are the following:

1. We propose and describe what is the first explicit proposal of a structured process to develop DApps, based on sound software engineering practices, and in particular on Agile principles.
2. We explicitly recognize that the development of smart contracts, and of the corresponding interaction apps which allow external actors to interact with the SCs, should be split in two flows. These flows are carried on concurrently and iteratively, after a common start, and are periodically integrated together.
3. We introduce a notation, augmenting some UML diagrams (Use Case, Sequence, and Class diagrams) in the context of Solidity language, to support SC design, accounting for their specificity. It is possible to use different UML extensions, to represent the design of SC written in other languages.
4. We provide checklists to fully support security and gas saving analysis. For the sake of brevity, security assessment of ABCDE method is thoroughly presented in another paper.

The proposed ABCDE method 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 Sec. 2 we describe the architecture of a DApp, and introduce the specific issues and practices needed for DApp development. In Sec. 3 we present the related work in the same, or similar fields. Sec. 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 Sec. 5, together with reporting on actual uses of ABCDE. Finally, Sec. 6 presents the conclusions and future work ideas.

2. Background

2.1. Decentralized applications

We define DApp as a software system that uses DLT, typically a blockchain, 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

[1] This is in particular required by the need for reaching consensus on the set of transactions to include into the blockchain.

enabling blockchain nodes, which needs different kinds of development practices, not the subject of this work.

A blockchain is an append-only, distributed data structure, managed by a set of connected nodes, each holding a copy of the blockchain, and able to execute SCs, programs residing in the blockchain itself. The blockchain state is changed through sending *transactions* to the network – in *public blockchains*, everyone can send a transaction, but only valid ones are processed. The valid transactions are 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 a single address, which is in turn associated to a private key. Only the owner of the private key can sign the transactions coming from an address, using asymmetric cryptography.

A transaction can transfer digital currency between addresses, 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.

Most present real applications of DApps and smart contracts are intended for the management of digital currencies or tokens, which 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 [13].

In this paper, we will use as a reference Ethereum, which is presently the most used blockchain to develop smart contracts on public blockchains [14,15]. Data on DApps running on permissioned blockchains are more difficult to find, because they refer to projects which are not publicly accessible, but Ethereum is very popular also for DApps running on permissioned blockchains. Open source DLTs such as Hyperledger and Corda are also widely used.

The Ethereum Virtual Machine (EVM), able to execute SC bytecode, runs on all nodes of the Ethereum blockchain [16]. In practice, the SCs are written in high-level languages (HLL). Nowadays, the most popular HLL for Ethereum is called Solidity.

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. Moreover, once a SC is deployed, it is in the blockchain forever – it cannot be modified or erased, though it can be forever disabled.

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. 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 (at least 10–15 s), and a greater amount of gas. In the latter case, the transaction is executed immediately.

A SC can receive and send Ethers, from and to another SC, or an address. A function which returns a value without changing the state of its SC is executed immediately by the EVM and costs nothing. This kind of function is called "*view*".

A typical DApp architecture is shown in Fig. 1. Here, the Ethereum blockchain is shown on the left, highlighting that a node is composed of its enabling software, of the EVM, and of the SC bytecode and its storage, recorded in the blockchain. The 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", is shown on the right. Its User Interface (UI) typically runs on a web browser. The server component stores data that cannot be stored in the blockchain, and performs business computations. In a non-trivial system, a DApp is typically composed of various SCs deployed on the blockchain.

ABCDE can be applied to develop DApps running both on public and permissioned blockchains.

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 [17]. 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

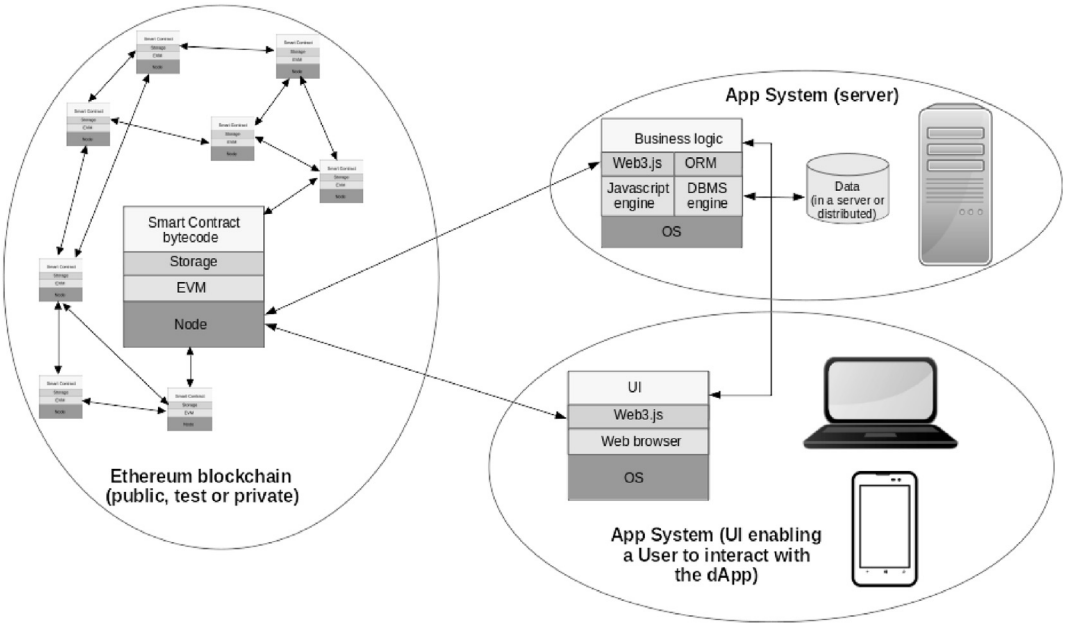

Fig. 1. A typical architecture of an Ethereum DApp application. The blockchain with its SCs is shown on the left, the App System on the right.

system requirements are highly available.

Other characteristics of DApp development is that DApps typically are not life-critical applications, though several among 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, because 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 the 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 [8]. 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) [11]. 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 [18]. 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 called "Product Owner". As described below in section 4.1, the proposed ABCDE method is based on Scrum, and is specifically focused on the software development process, and not on specific design or programming practices.

Besides Scrum process, there are other agile practices that are well suited to DApp development. In the followings, we list and shortly describe agile practices that we suggest to use in ABCDE process, and how they can be applied to smart contract development.

- *Test Driven Development (TDD)*: this practice prescribes writing the tests before the code [19], using an automated test suite that can be run whenever needed. For the App System, one can use one of the many existing testing frameworks. For SCs written in Solidity, at the moment the most popular testing environment is Truffle [20]. Note that performing automated testing on SCs is not trivial, because tests need to be run on the blockchain, which is a separate entity from the testing environment itself. This is similar to testing the interface of a database. Also, in order to test the software interacting with the SCs, we need a "mock object" able to simulate the blockchain, if the required SCs are not yet implemented, and/or if we need to improve the speed of testing.
- *Continuous Integration*: merging all developers 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, and the following one, requires the use of a version control system helping integration and versioning, as well as of an automated test suite, to assess the absence of undesirable side-effects.
- *Collective Code Ownership*: every developer is allowed 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. As regards smart contract development, team members modifying code written by other members should be very careful not to infringe security and gas optimization provisions. Note that often the team members expert in SC development differ from those expert in App System, so their spheres of influence should remain separate.
- *Refactoring*: this practice too needs to have an automated test suite, that can be run when the refactoring is made, to assess the absence of unwanted side effects [21]. 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 can obviously greatly benefit DApp development and its use is therefore strongly encouraged in ABCDE.
- *Coding Standards*: 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. This would greatly facilitate code understanding, and ease subsequent maintenance activities.
- *Pair Programming (PP)*: using ABCDE, 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 subsection, 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 many 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 Sec. 4.4.

3. Related work

3.1. Software engineering for DApp development

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. [7]. They highlight "*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*" [7]. They also suggest the adaptation of existing design notations, such as UML, the Unified Modelling Language [22] 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 [23]. 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 Ref. [24].

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 [25].

Fridgen et al. propose an approach for eliciting use cases in the context of blockchain-based applications, applying action design research method. Their method is evaluated in four distinct case studies regarding banking, insurance, automotive and construction [26].

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 [27].

A paper by Beller and Hejderup [28] is worth mentioning, though it does not really advocate to use SE practices to develop blockchain-based 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. [29].

Chakraborty et al. used an online survey to get 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 threats*" [9]. 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*" [10]. 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.

3.2. Security for DApps

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 smart contract security, summarizing the main security attacks against SCs, their key vulnerabilities, the security analysis methods and tools [30]. 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 [31]. 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.

The works on SC security consider in depth the various kinds of attacks and vulnerabilities of DApps, and how to find and mitigate them. However, they typically do not take an overall approach to secure software development life cycle. This is a relatively recent field, whose forefront representatives are Microsoft's Security Development Life cycle (SDL), OWASP's Comprehensive, Lightweight Application Security Process (CLASP) and McGraw' Touchpoints [32]. Though secure software development mostly prefers waterfall-like methodologies, it can be performed also with agile processes [33].

3.3. Domain-specific UML additions

Various papers have been published to suggest upgrades of Unified Modeling Language [22] 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 [34].

Baresi et al. [35] extend and customize UML with web design concepts borrowed from the Hypermedia Design Model. Hypermedia elements are described through appropriate UML stereotypes.

Rocha and Ducasse [36] 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. Rationale and motivation

Our approach, ABCDE, takes into account the substantial difference between developing traditional software (the App System) and developing smart contracts, and separates the two activities. For both developments, ABCDE takes advantage of 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. This ruled out the use of plan-driven methods such as waterfall, and iterative-incremental methods relying on longer iterations.

ABCDE is an agile method based on Scrum [11], due to Scrum's simplicity, its popularity – Scrum is by far the most used software development method [37] – and also due to the specific experience of the authors in studying and applying Scrum [38]. In Scrum, a subset of USs are implemented at each iteration. Also a Lean-Kanban approach would be feasible, implementing the USs in a continuous flow, with the *work in progress* controlled by the Kanban board [39]. In this case, the board should show, in different "lanes", the USs of both the SC system and the App System. However, because many DApp development projects are new, and thus being built from scratch and with a dedicated team, we deemed that Scrum is more suited than Kanban – which is instead very suited for teams working concurrently on multiple projects. From Scrum, we kept the requirement gathering with user stories, the iterative-incremental approach, the key roles, and the meetings (sprint planning, daily Scrum, sprint review, and sprint retrospective). The main differences with Scrum are:

- the separation of development activities in two flows, each performed iteratively, with integration activities every 2–3 iterations;
- explicitation of the activities that must be performed to design, develop, test and integrate smart contracts and DApp system – this is not included in Scrum;
- emphasis on documenting the smart contracts using formal diagrams, to help development, security assessment, and maintenance – these diagrams are not intended to be exhaustive, but are not required in Scrum;
- specific activities related to security assessment and gas optimization.

To document in a structured way the smart contracts, we found very useful some UML diagrams, properly modified, which are described in section 4.3. We used UML because it is by far the most used modeling language in software engineering, and is provided of the "stereotype" construct which enables to add easily the required features to the diagrams. UML provides standard diagrams to effectively model both the data structure of smart contracts (class diagram), and the exchange of messages between the entities of the DApp system (sequence diagram).

Eventually, our experience in software quality assessment, made us appreciate the systematic use of patterns and checklists. These tools greatly help developers to proceed in a structured and systematic way. For this reason, we used this approach for security assessment and gas optimization, starting from an accurate literature investigation on the subject. Sections 4.4 and 4.5 describe in detail these components of ABCDE.

4.2. The process

As written before, ABCDE is based on Scrum. The key roles of Scrum, and consequently of ABCDE, are Scrum Master (which might be called *ABCDE Master*), Product Owner and Team. These roles are well known, so we will not describe them in detail.

The steps of the proposed ABCDE design method, which is currently focused on Ethereum blockchain and Solidity language, are shown in Fig. 2. Note that most steps are in fact performed many times, because the approach is iterative and incremental. In the figure, the pink circles

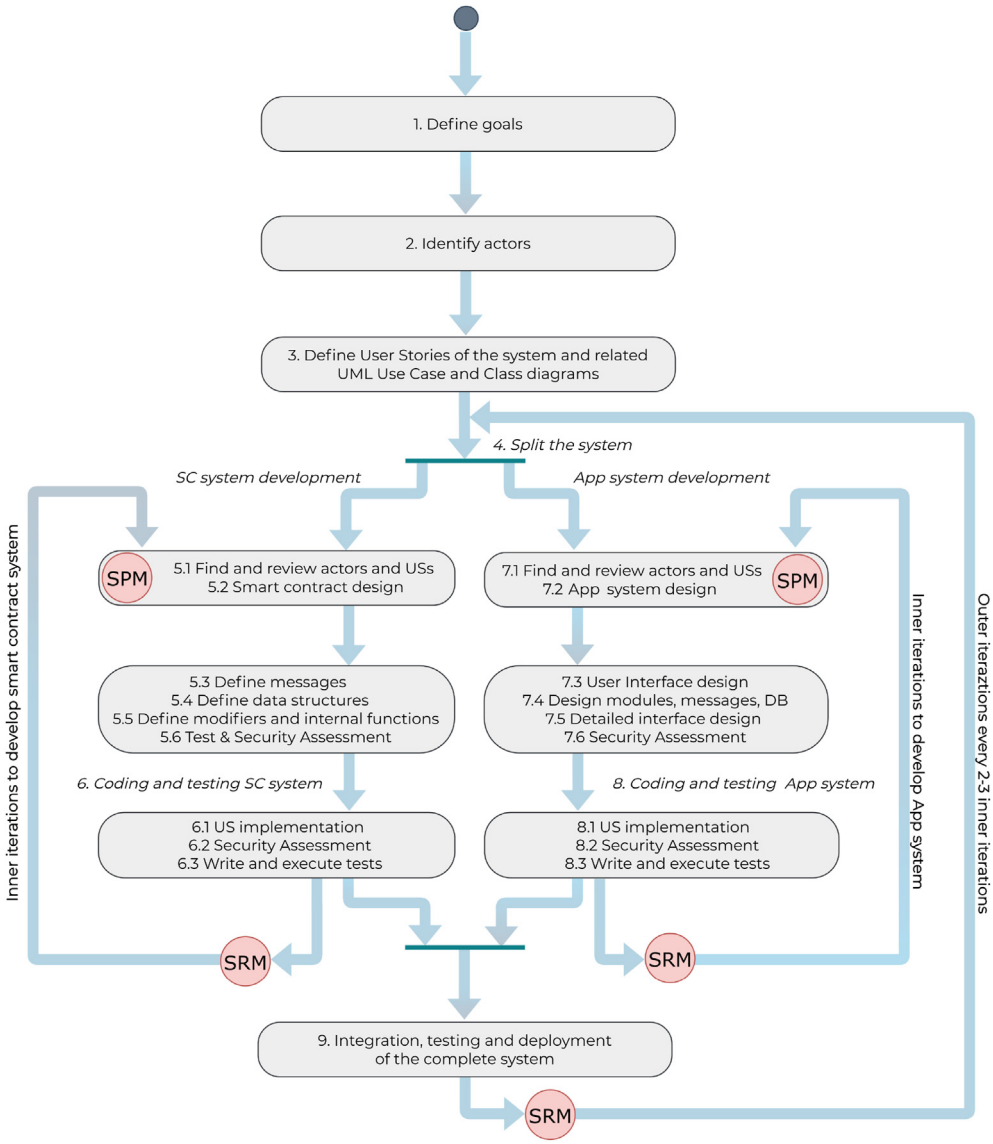

Fig. 2. The proposed ABCDE process; the circles represent the Scrum meetings.

represent sprint planning meetings (SPM) held at the beginning of each sprint (iteration), and sprint review meetings (SRM) held at the end of sprints. Daily scrums (stand-up meeting held each day) and retrospective meetings are not reported in Fig. 2.

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 [11], but here the goal is 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) [18], to be able to follow the classical agile approach for project management, used in Scrum [11]. 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 here. 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, for instance by applying the decision framework proposed by Scriber [41], the following steps are taken.
4. **Divide the system in two subsystems.**
   - The smart contracts running on the blockchain (Steps 5–6).
   - 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 smart contract executes a function (Steps 7–8).

At this point, an architecture of the whole system should be drafted, highlighting what data should be put on-chain and what should be placed out-of-chain. The guideline is that SCs should manage the data and processing that need to be transparent and immutable for the DApp to be trusted by its actors. This includes the management of actors' identity.

All other data, processing and user interfaces should be managed out-of-chain. In the case of data which must be trusted, but cannot be stored in the blockchain due to its transparency on-chain, data privacy can be achieved using the "Off-Chain Data Storage" pattern [23]. Leakage of

transaction volume and parties involved might still be possible, and can be avoided by further obfuscating techniques.

5. **Design of the smart contracts.** 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 Ref. [9]. The activity is performed through iterations that include coding and delivering increments of SCs, which are the USs chosen for each iteration. It is divided in sub-steps, which we explicitly consider and list following a logical sequencing (but which should not necessarily be performed in a "waterfall" sequence). These sub-steps, as well all the sub-steps of the following main steps, derive from our experience in smart contract and DApp development, and from discussions had with many DApp developers. They 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 user story defined in this step, define also the related acceptance test(s).
   - 5.2 Define broadly the SCs composing the SC subsystem. For each SC, state its responsibilities to store information and to perform computations, and the related collaborations with other SCs. For non-trivial systems, you will typically need various interacting SCs. Also consider the use of inheritance for abstracting common features of SCs. Describe in detail the collaborations with external SCs, including libraries. UML class diagrams with proper additions will be used, as shown later in Sec. 4.3.
   - 5.3 Define the flow of messages and Ether transfers among SCs, external SCs and the App System. Use augmented UML sequence diagrams to document these interactions, if they are non-trivial (see Sec. 4.3). If needed, define the state changes of SCs using UML statecharts.
   - 5.4 Define in detail the data structure of each SC, its external interface (Application Binary Interface, ABI) and the relevant events that can be raised by it.
   - 5.5 Define the internal, private functions and the modifiers – special functions that usually test the preconditions needed before a 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. Sec. 4.4 will describe 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. The coding and testing activities are:
   - 6.1. Incrementally write and test the SCs. Owing to the strict security requirements, typically this activity cannot be performed in a strict incremental way, just implementing one user story 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 and gas optimization of the code written for the increment (see Tables reported in Refs. [42] and in section 4.5).
   - 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 [20]. 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 can raise events. 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, setting up and running one or more nodes, 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 application. We suggest to perform UI design using a well known standard approach, such as Usage-Centered Design [43] or Interaction Design [44].
   - 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 through the "Off-Chain Data Storage" pattern [23]. 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. This design activity is not performed up-front, but through iterations that include coding and delivering increments of the App System, implementing USs chosen for the iteration. Due to the strict 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 also 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, using the same approach of SCs development (Scrum or Lean-Kanban). 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. 2. If a continuous-flow, Lean-Kanban approach is performed, the integration should happen at the completion of a given set of USs, in both branches; it will be activated by a specific user story put on the Kanban board. The activities happening in parallel are:
   - 8.1 Incrementally implement the USs of App System. This step belongs to the "right flow" of ABCDE (see Fig. 2), and does not differ from the implementation of a web application.
   - 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.** To integrate SCs and App System, the overall systems built up to that moment must be deployed into a local or a testnet blockchain, and integration tests must be run to check whether all the components interact together as expected (e.g. events raised by SCs are collected by the App System, messages sent by the App System activate blockchain transactions that are validated and correctly executed, and so on).

4.3. UML diagrams for modeling SCs

As written before, the most popular blockchain for DApp development is presently Ethereum, and the most used language is Solidity [45]. This language is object-oriented (OOPL) because smart contracts are defined similarly to classes – they have internal variables, and public and private functions able to access these variables. However, Solidity has no *true* classes, but only smart contracts. Each SC can inherit from one or more other SCs. With respect to a standard OOPL, Solidity adds specific concepts like events and modifiers, and exhibits strong 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 0.7. 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. Note that 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 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 an 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 graphically show 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 specificities, and their description.

Table 1
Additions to UML class diagram (stereotypes).

Stereotype	Position	Description
«contract»	Class symbol – upper compartment	Denotes a SC. May also be «abstract contract»
«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. There is no bottom compartment (holding operations).
«struct»	ditto	A record, able to hold heterogeneous data. The fields are listed in the middle compartment. There is no bottom compartment.
«event»	Class symbol, middle compartment	An event that can be raised by a SC's function, signaling 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	The multiple variable, or the 1:n relationship, is implemented using an array.
«mapping»	ditto	The multiple variable, or the 1:n relationship, is implemented using a generic mapping.
«mapping [*address*]»	ditto	A multiple variable, or the 1:n relationship, which is implemented using a mapping from an Ethereum address to the value.
«mapping [*uint*]»	ditto	A multiple variable, or the 1:n relationship which is implemented using a mapping from an unsigned integer to the value.

A special kind of transaction is used to create a SC, after its source code has been compiled to bytecode. The other two kinds of transactions are the transfer of Ethers, and the invocation of a function on an existing SC (message).

To address the need to manage complex data, Solidity has the "struct" construct. The relationships among SCs and/or structs can be effectively captured by a UML class diagram:

- the multiple inheritance among SCs is the same as with classes;
- when a SC sends a message to another SCs, they can be linked using an association (if they are logically associated), or a dependence;
- structs and enums can be included in the data structure of a SC, and this relationship is modeled using a composition.

A specific concept of Solidity are *events*, raised when something relevant happens, which can be caught by observer programs. Remember that SCs cannot directly invoke functions of external systems, and thus events are a mean for SCs to communicate with the external world.

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 an association, aggregation or composition. The array is an ordered set of values, indexed by their position, as in most computer languages. 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.

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

A characteristic of Ethereum is that messages sent to a SC through a transaction take time (typically 15–20 s or more) to be answered. However, if a message is sent to another SC during the execution of a function of a given SC (Contract Internal Transaction), 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 response time, security and gas consumption, we introduced the stereotypes «*trans*-msg» and «internal-msg» tagging the message calls sent through a transaction, and directly by a SC, respectively.

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, 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».

Finally, the «fallback» stereotype tags the homonymous special function of each SC, which is called whenever a message is not matched, or an Ether transfer fails. This function implements recovery procedures, and is particularly critical for security.

Our Sequence Diagrams represent the message exchange among external actors and SCs, all called *participants*, in a given scenario. The messages between external actors follow the usual UML notation. An external actor, however, can also send Ethers to another. Our notation allows the use of standard frames and fragments, such as "alt", "opt" for condition testing, "loop" for loops and "par" for fragments running in parallel. Table 2 reports the stereotypes we introduced in UML Sequence diagrams to identify the participants sending messages from their unique address, and the kinds of messages they exchange.

4.4. Security assessment for smart contracts

Assessing SC and, in general, DApp security is a difficult task, due to the complex and new architecture of DApps. A sound method for DApp development, however, cannot overlook security. Since DApp security assessment covers a large amount of concepts and guidelines, we wrote a specific paper on the subject written by our research group, which we refer to Ref. [42]. Here we briefly recall the key advice of this paper. Following a secure software development lifecycle approach, ABCDE does not limit security assessment to testing, or to a specific phase performed after development, but it introduces security assurance practices in all three phases of design, coding and testing. Moreover, ABCDE stresses that 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-revealing code, use of simple tools. All these practices surely help security. However, Agile means also incremental development, where USs are continuously completed, added to the current working system and tested. This greatly helps productivity, but might be at the expense of security, because there is the risk that these continuous additions may introduce unwanted side effects, and even security breaches.

A good starting point to focus on security are the Top 10 Proactive Controls of OWASP organization [46]. Those most relevant for DApp

Table 2
The stereotypes added to UML Sequence diagrams.

Stereotype	Position	Description
«person»	Participant box	A human role who posts transactions using a wallet or an 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.
«wallet»	ditto	An Ethereum wallet, holding the private keys to access addresses, able to send transactions and to interface with its owner.
«*trans*-msg»	Message	The message is sent using an Ethereum transaction.
«internal-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 drawn also as a stand-alone message.

Table 3
Main gas saving patterns.

Name	Description	Ref.
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.	[53]
Eternal Storage	If a SC must hold several data, use a separate SC acting as a storage to it. A new version of the original SC can use the same storage SC as its predecessor, after it has been linked to it.	[54]
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.	[52, 55]
Pack 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. Use datatype *bytes32* rather than *bytes* or *string*, if possible.	[52, 55]
Delete variables	If you don't need a variable anymore, delete it using the *delete* keyword. In Ethereum, you get a gas refund for freeing up storage space.	[52, 55]
Do not initialize variables	All variables are initialized to zeroes at no cost. Initialize them only if non-zero.	[52, 55]
Use Mappings	To manage lists of data, use mappings with integer key and not arrays. This is known to save blockchain space.	[52, 55]
Execution Paths	Thoroughly examine all possible execution paths, looking for code whose execution can be spared.	[52, 55]
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.	[52, 55]
Limit modifiers	The code of modifiers is "inlined" inside the modified function, thus costing gas. Internal functions, on the other hand, are called as separate functions. They save a lot of redundant bytecode in deployment, if used more than once.	[52, 55]
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.	[52, 55]
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.	[55]

Table 4
Survey on ABCDE usage.

Nr.	Question	Mean	St. Dev.	Min	Max
1	Years of sw. development experience	8	9.21	2	35
2	Years of agile sw. development experience	4.5	4.54	0	15
3	Years of DApp development experience	2.6	1.70	7 months	5
4	Number of completed DApp projects	2.5	2.44	1	9
5	Number of ongoing DApp projects	1.3	0.91	0	3
6	Number of DApp projects performed using ABCDE	1.9	1.14	1	5
7	ABCDE is overall useful	4.3	0.61	3	5
8	ABCDE is useful for requirements elicitation	4.4	0.65	3	5
9	ABCDE is useful in system design using UML diagrams	4.5	0.66	3	5
10	ABCDE is useful for its iterative/incremental approach	4.0	0.68	3	5
11	ABCDE is useful in security analysis	3.8	0.70	3	5
12	ABCDE is useful in optimization of gas consumption	3.5	0.85	2	5
13	ABCDE is useful in the testing phase	3.8	0.97	3	5
14	ABCDE is useful for integrating SC and DApp systems	3.7	0.95	2	5
15	ABCDE is easy to use	4.4	0.50	4	5

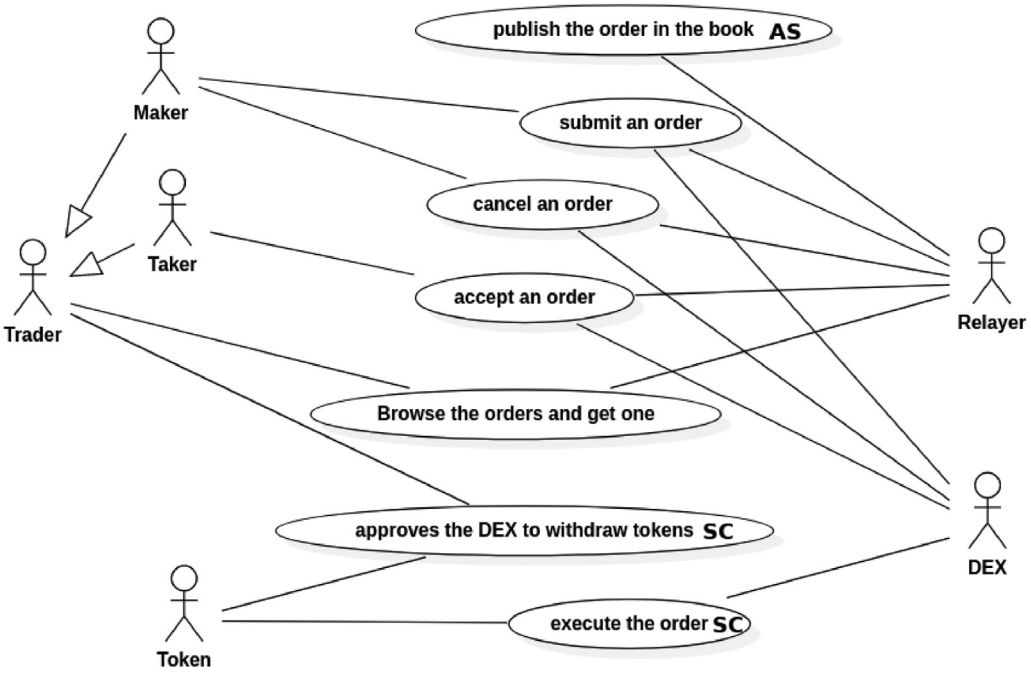

Fig. 3. The User Stories of the DEX system specification.

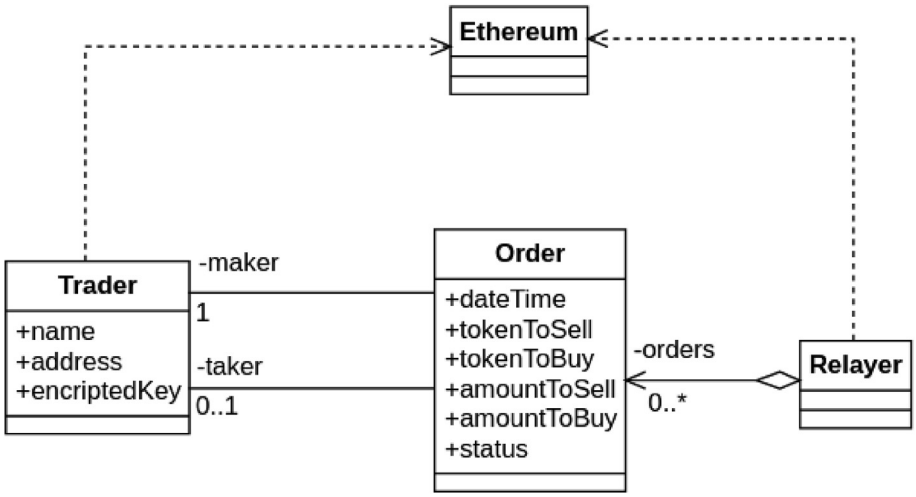

Fig. 4. The standard UML class diagram derived from the USs.

security are: An explicit definition of the security requirements needed (C1); Reuse of software that is security-hardened (C2); Systematic input validation (C3); Sound management of identities and related access controls (C6, C7); Data protection (C8); Handling all errors and exceptions (C10).

General guidelines specific to SC security, which complement OWASP ones, are reported in Ref. [47], section: "General Philosophy". They are:

1. *Prepare for failur*e. Be able to respond to errors, also in the context of SCs, which cannot be changed once deployed.
2. *Rollout carefully*. Try your best to catch and fix the bugs before the SC is fully released.
3. *Keep smart contracts simple.* 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.* Your previous programming experience is also applicable to SC programming, but there are several specific pitfalls to be aware of.

Our approach includes some security checklists, to be performed during and after design, coding and testing phases. The aim is to verify that all security patterns and practices concerning known problems are applied. We also include a checklist for gas optimization, which is crucial not only for saving money, but also for sparing DOS attacks and avoiding unwanted SC execution abort for running out of gas.

ABCDE approach is iterative and incremental. When developing DApps which manage real value, like digital money or tokens, it's important that all stakeholders (or blockchain governance body) agree on the smart contract logic and implementation. This means that all stakeholders should be involved in each iteration, or at least in the iterations leading to the integration of SC and App systems. In the case this is not possible, one has to anticipate the detailed design and coding of the

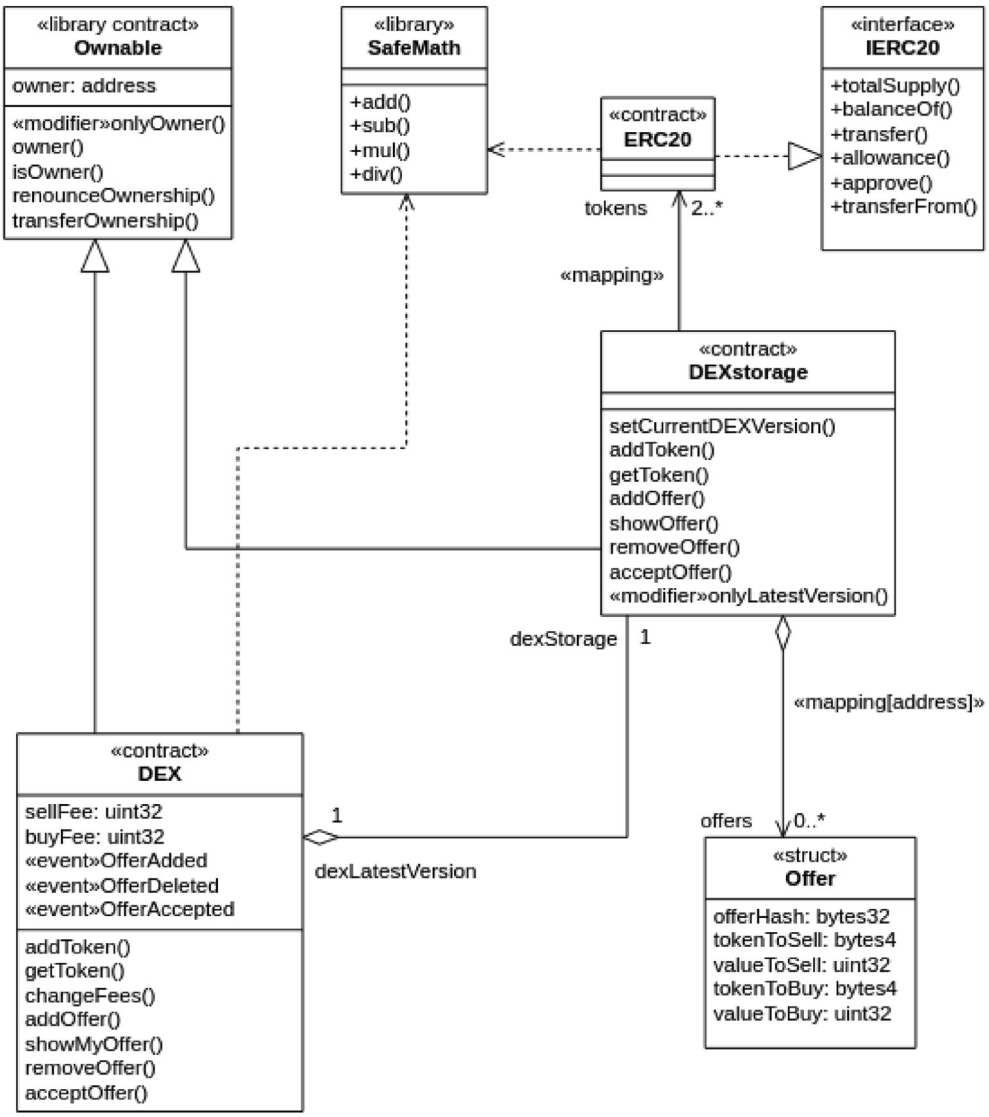

Fig. 5. The modified UML class diagram, showing the structure of the required smart contracts of the DEX system.

key contracts at a given set of iterations – involving all stakeholders – and then give the Product Owner the responsibility of checking and taking note of all the changes.

For a detailed description of security patterns and checklists, please refer to paper [42]. In all phases, 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. A spreadsheet file with the checklists is available at the following link: http://tiny.cc/security_checklist. Here we'll outline the main issues and patterns for each development phase.

4.4.1. *Security in the design phase*

During smart contract design, you must think more strategically, and apply patterns and checks regarding the architecture and general modeling of the SCs.

Here, you must decide if and how to apply decoupling and fail-safe patterns, such as Proxy [48] and Check-Effect-Interactions [49]. Minimization of dependencies, and careful planning of reuse through inheritance and external libraries is another activity typically performed in the design phase [47].

You should also decide how to manage authorizations to the use of the system, and how to avoid race conditions due to wrong assumptions on system time and transaction ordering [50]. You should carefully plan Ether management, if your SCs have to hold, receive and deliver Ethers. To this purpose, it is wise to limit amounts and frequency of Ether withdrawals, and use a "pull" approach to it [47].

4.4.2. *Security in the coding phase*

When coding smart contracts, one major class of potential issues derives from code performing "external calls'', calling functions of other SCs during their execution. In Ethereum, a SC can call another SC's public function. The call can be recursive, so the called SC can in turn perform an external call, and so on. So, external calls must be treated like calls to 'untrusted' software, and should be avoided or minimized. In fact, malicious code could be introduced somewhere in a SC belonging to this path. It is true that all external SCs are already present in the blockchain, and thus are immutable. However:

- if their code is not thoroughly checked by a competent professional, a SC might not work as intended;
- if the called SC makes use of the *Proxy* pattern, it can be changed by its author;
- in complex DApps, to avoid rewriting of the whole system in the case of a change, mechanisms to dynamically change the address of the called SC are typically used;

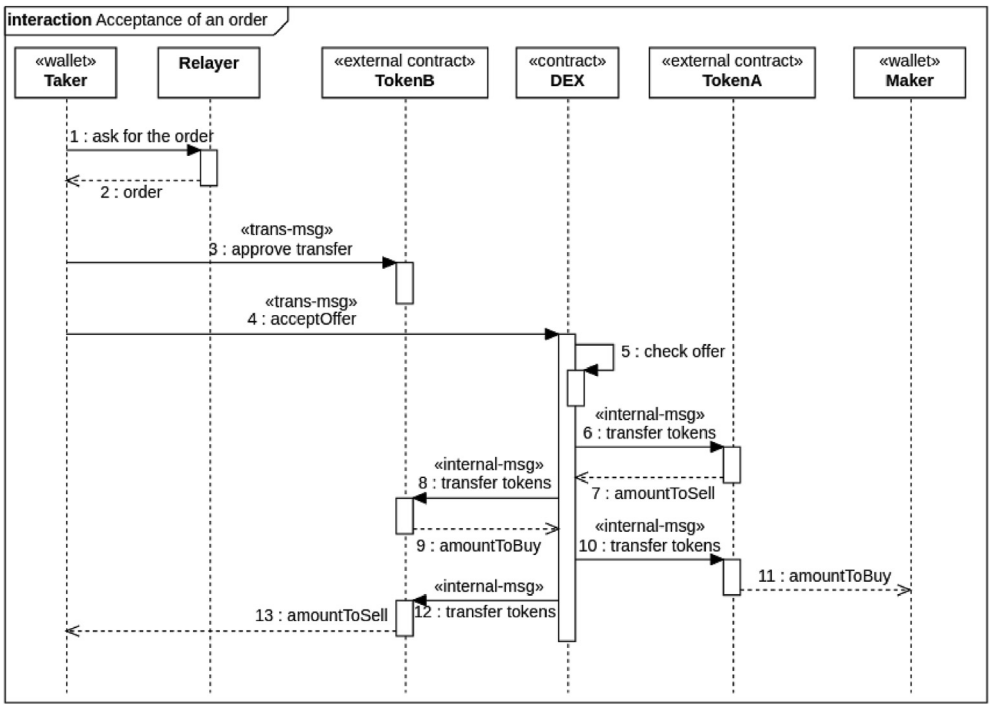

Fig. 6. The UML sequence diagram showing a Taker accepting an offer and sending it to the DEX for execution.

- last but not least, an attacker can exploit the fact that a message sent by the target SC can activate the fallback function of the attacking SC – this 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 with security implications is the use of *assert()*, *require()* and *revert()* guard functions for error handling. These functions are the subject of security pattern "Guard Check" [51].

assert() should be used to validate state after making changes, and to check for data consistency. When an *assert()* statement fails, something very wrong happened and you need to fix the code. *require()* should be used to validate data: user inputs, state conditions preceding an execution, or the response of an external call. *revert()* is used to handle the same type of cases as *require()*, but with more complex logic [47].

4.4.3. Security in the testing phase

The most important tool to achieve security and correctness, however, is to apply thorough, automated tests. This is even more crucial when writing smart contracts, because it is difficult or impossible to update a SC.

ABCDE does not prescribe the use of specific 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 Ref. [20], section: Testing Your Contracts.

4.5. Gas optimization

Besides security, another important factor of smart contracts 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 3 shows some specific patterns that can be used to save gas, as more thoroughly discussed in Ref. [52].

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 [23]). 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.

5. Experimental validation

The development process which later was named ABCDE was first devised in 2018 [12], and since then it has been used in several projects 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 manage temporary job contracts; a couple of systems to trace the provenance of foods (one of which developed using Hyperledger technology) [56]; two voting systems, one managing voting in firm shareholders' and board of directors meetings, the other for anonymous voting; a system to manage energy exchange in local networks of electricity producers and consumers; a system to automate agile software development [29]; a system to notarize and to manage incentives for check-up visits.

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. In Table 4 we report the results of a survey conducted among 14 developers and graduate students who used ABCDE on at least one DApp project. The scores regarding the features of ABCDE method (questions 7–15) span from 1 (not useful at all) to 5 (very useful); a neutral opinion corresponds to a score of 3.

As you can see, the experience in software and DApp development varies greatly. The average number of DApp projects performed by respondents is fairly high (almost 4 projects each, of which 2.5 closed and 1.3 ongoing). The overall satisfaction for ABCDE method is quite high, as well as its ease of use. The strengths of the method are especially in the analysis and design phases, whereas it is less appreciated (but still above sufficiency) in gas optimization, and final integration.

The respondents gave also several suggestions on possible improvements to ABCDE, some of which are reported in the final section of this paper.

5.1. Building an example DApp

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 0× protocol project, the subject of a successful ICO held in 2017. The specification of the DEX can be found in the 0× Whitepaper [57]. 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), and the signing of the offers by traders. In our example, a trading offer is simply posted to the DEX, and who wish to accept it simply sends a transaction to the DEX. The guarantee against frauds are the transparency of the underlying SC, and the hash signature of each offer, which guarantees against fraudulent changes of the offer after it is accepted.

5.2. The first steps of ABCDE

For the sake of brevity, we will not present the App System coding phase, and the system integration phase (phases 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 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 web 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 protocol.
3. **User Stories**. Fig. 3 shows the actors and the user stories 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 but, given the simplicity of the example, they are self-explaining. In Fig. 4 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 shown in Fig. 3. In short, the system just deals with orders, posted by makers and later accepted by takers, who are kinds of traders. The Relayer is the service which publishes the offers and makes possible the exchange of tokens. Both traders and relayer access the smart contracts implementing the tokens in the blockchain.
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 USs of the external app subsystem are the same of those reported in Fig. 3, except those related with direct interaction with the blockchain, tagged “SC” in the diagram of Fig. 3. Also the USs of the blockchain subsystem are the same of those reported in Fig. 3, but that tagged “AS".
5. **Design of the SC subsystem**. The SC system is quite simple, and mainly involves the “DEX” SC, which interacts with the SCs managing the supported tokens to exchange. The data managed by the DEX are the trading fees, the list of supported tokens, and the list of the offers. To hold these lists we use the “Eternal Storage” pattern, consisting in storing them in an external SC, so that possible changes to the DEX can be managed with no need to store again all these data [54]. Conversely, the use of the Proxy pattern [53] is excluded, because the guarantee that the DEX works properly is given by inspecting its source code. Giving the DEX’s owner the ability to change the DEX, leaving the Proxy unchanged, would void this guarantee. We report in Fig. 5 the UML class diagrams showing the SCs of the system. This diagram shows some of the specific stereotypes used to document an SC system, as described in Sec. 4.3.

The entities shown in the top of the diagram are standard library contract “Ownable”, library “SafeMath” and “ERC20” token interface, used in most contracts dealing with tokens. “ERC20” contract represent at least two token contracts active on Ethereum blockchain and managed by our DEX. The DEXStorage contract holds zero or more “Offer” records, and is linked to the related DEX contracts. Modifiers and events are shown in the corresponding contracts.

Fig. 6 shows a UML sequence diagram representing the interactions among most Actors of the systems, when a Taker accepts – through her/his wallet – an order seen in the Relayer’s book, and sends it to the DEX for execution, including the messages exchanged among the SCs. Basically, the Taker approves, the transfer to the DEX of the tokens to give to the Maker, plus the DEX fee; after that, the DEX cashes the tokens from both Maker and Taker, keeps the fees and gives back the proper tokens to both Traders’ wallets.

6. **Coding of the SC subsystem**. The developed SCs make use of existing library SCs, namely “OnlyOwner” to manage the ownership of a SC, and “SafeMath” to avoid over- and under-flow errors. They also refer to SCs already deployed on the blockchain and implementing the ERC20 standard interface for managing tokens.

The contract “DEXstorage” holds and manages the mapping “tokens” having as key the supported token symbol (4 characters) and as value the token address, and the mapping “offers” having as key the Maker’s address and as value the details of the offer (symbols and quantities of the tokens to sell and buy, and their hash digest of these data). The owner of DEXstorage is its creator, who is able to change the address of the DEX, stored in “dexLatestVersion” variable. All other DEXstorage operations can be performed only by the DEX. This is ensured by using “onlyLatestVersion” modifier. A part of DEXstorage contract follows, with the Offer definition:

```
pragma solidity ^0.7.0;
import "Ownable.sol";

contract DEXstorage is Ownable {
    address dexLatestVersion;  // Address of DEX contract using this storage.
    struct Offer {
        bytes32 offerHash;    // Hash of offer's parameters.
        bytes4 tokenToSell;  // Token to sell symbol.
        bytes4 tokenToBuy;   // Token to buy symbol.
        uint256 amountToSell; // Amount to sell.
        uint256 amountToBuy;  // Amount to buy.
 }
 mapping (bytes4 => ERC20) public tokens;
 mapping (address => Offer) public offers;

 modifier onlyLatestVersion() {  // Only  DEX contract using this storage.
    require(msg.sender == dexLatestVersion);  _; }

 function setCurrentDEXVersion(address _dex) external onlyOwner {
     dexLatestVersion = _dex; }

 function addOffer(address _maker, bytes4 _symbolToSell, bytes4 _symbolToBuy,
                   uint256 _toSell, uint256 _toBuy)
                                            external onlyLatestVersion {
     offers[_maker] = Offer(
         keccak256(abi.encodePacked(_symbolToSell, _symbolToBuy, _toSell, _toBuy)),
         _symbolToSell, _symbolToBuy, _toSell, _toBuy); }

 function getOffer(address _maker) external view
                    returns(bytes32, bytes32, bytes32, uint256, uint256) {
     Offer memory _makerOffer = offers[_maker];
     return (_makerOffer.offerHash, _makerOffer.tokenToSell, _makerOffer.tokenToBuy,
             _makerOffer.amountToSell, _makerOffer.amountToBuy);
 }

 function removeOffer(address _maker) external onlyLatestVersion {
     Offer memory nullOffer;
     offers[_maker] = nullOffer;
 }
}
```

The contract "DEX" implements the decentralized exchange. It allows its owner to add the supported tokens, and then the Maker to add offers. Each Maker can have just one offer active at a given time. Before posting the offer, the Maker must approve the DEX address to withdraw from the SC managing the token to sell the offered amount, plus the selling fee. The function "addOffer" is shown below:

```
 pragma solidity ^0.7.0;

import "SafeMath.sol";
import "ERC20.sol";
import "DEXstorage.sol";

contract DEX is Ownable {
    using SafeMath for uint256;
    uint256 public sellFee;         // Constant amount of fee paid by makers.
    uint256 public buyFee;          // Constant amount of fee paid by takers.
    DEXstorage private dexStorage;  // DEXstorage implementation reference.
...
    function addOffer(bytes32 _symbolToSell,  bytes32 _symbolToBuy,
                    uint256 _amountToSell, uint256 _amountToBuy) external {
        require(address(dexStorage.tokens(_symbolToSell)) != address(0));
        require(address(dexStorage.tokens(_symbolToBuy)) != address(0));
        require(dexStorage.tokens(_symbolToSell).allowance(msg.sender,
                address(this)) >= (_amountToSell.add(sellFee)),
                                            "Amount to sell exceeds allowance.");
        dexStorage.addOffer(msg.sender, _symbolToSell, _symbolToBuy,
                                        _amountToSell, _amountToBuy);
    }
...
}
```

A Taker can accept the Maker's offer. Of course, also the Taker must previously approve the DEX address to withdraw from the SC managing the token to buy the offered amount, plus the buying fee. The "acceptOffer" code is shown below:

```
function acceptOffer(address _maker, bytes32 _offerHash) external {
    bytes32 offerHash;
    bytes4 tokenToSell;    bytes4 tokenToBuy;
    uint256 amountToSell;  uint256 amountToBuy;
    (offerHash, tokenToSell, tokenToBuy, amountToSell, amountToBuy) =
                                dexStorage.getOffer(_maker);
    require(offerHash == _offerHash, "The offer hash doesn't match.");
    require(dexStorage.tokens(tokenToSell).transferFrom(_maker, address(this),
                amountToSell + sellFee),"Transfer from Maker to DEX unsuccessful.");
    require(dexStorage.tokens(tokenToBuy).transferFrom(msg.sender, address(this),
                amountToBuy + buyFee), "Transfer from Taker to DEX unsuccessful.");

     require(dexStorage.tokens(tokenToBuy).transfer(_maker, amountToBuy),
                                  "Transfer from DEX to Maker unsuccessful");
     require(dexStorage.tokens(tokenToSell).transfer(msg.sender, amountToSell),
                                  "Transfer from DEX to Taker unsuccessful");
     dexStorage.removeOffer(_maker); }
```

As in the "addOffer" function, most actions are performed inside a "require" clause, meaning that if the action aborts the whole computation aborts, the blockchain state does not change, and the remaining gas is sent back to the caller.

5.3. Security assessment

Although the DEX system is quite simple, it has strict security requirements, because it manages tokens, which can usually be exchanged with real money. In the followings, we follow the security checklist to be applied in design phase (8 items) and coding phase (18 items), as reported in Refs. [42]. For the sake of brevity, we will not report 18 checks which are not relevant for our case study.

Limit the amount of ether: we check that all money transfers are performed through explicit withdrawals made by the beneficiary address.

Transaction Ordering: we added the hashing check of orders because, lacking it, a fraudulent Maker could post a very favorable offer; then the Maker might detect a Taker's transaction to accept the offer, and quickly post a faster transaction (with much higher gas value) changing

the offer in an unfavorable way. The hash signature of the offer solves the problem.

Use trustworthy dependencies: we use only standard and very proven libraries, such as "SafeMath.sol" and "ERC20. sol".

Beware of re-entrancy: using the Check-effect-interaction: all relevant actions are preceded by checks. No call to other SCs are made which might trigger reentrancy issues.

Embed addresses to grant **permissions**: most functions able to modify the SC storage can be called only by the owner, or by the DEX in the case of DEXstorage. The only functions callable by external addresses are those used by the Maker to post an offer, and by Taker to accept an offer.

Use platform related standards: the only external SC called are ERC20 tokens, which are a proven Ethereum standard.

Prevent overflow and underflow and Beware of rounding errors: the (few) relevant computations are performed using "SafeMath" library, which protects from these kinds of errors.

Validate inputs to external and public functions: not only all relevant actions are preceded by checks, but all actions are performed including them inside a "require" clause. If a guard is not satisfied, the whole action aborts.

5.4. Gas optimization

The minimization of gas consumption might be considered part of performance optimization of any software solution. With DApps, the blockchain itself is the main performance bottleneck, because external transactions take time to be processed and accepted, even in a permissioned blockchain. With Ethereum, optimizing the performance of smart contracts corresponds to minimize gas consumption, which is linked also to the number of bytecode instructions executed.

The presented code follows the main gas optimization patterns, as presented in Table 3. In particular, the following patterns are relevant:

Proxy Delegate: We did not use Proxy Delegate pattern for the reason explained in Step 5 of section 4.2.

Eternal Storage: this pattern corresponds to the use of "DEXStorage" contract, which would allow to save a lot of gas in the case the DEX contract needs to be upgraded. In fact, if the new DEX version uses the same DEXstorage of the previous one, all tokens and offers can be reused. In this case, however, the Makers must be warned to change the DEX address in their approval to withdraw tokens.

Pack variables: we packed the token symbol byte arrays is in the "Offer" structure.

Do not initialize variables: we did not initialize variables with default values.

Use Mappings: all the needed collections in our contracts are implemented using mappings.

Execution paths: our functions do not perform heavy computations. However we checked all possible execution paths to make sure they are minimized.

Limit external calls: our functions are typically declared as "external", which are cheaper than "public" functions.

Limit modifiers: at most one modifier is used per function.

5.5. Writing automated tests

Each non-trivial function of the contracts written so far must be provided of Unit and Acceptance Tests. Truffle framework [20] makes available two methods for testing Ethereum smart contracts: Solidity test and JavaScript test. For the sake of brevity, here we report just a fragment of the Solidity unit test written to verify "addToken()" and "addOffer()" functions of "DEXstorage" contract:

```
pragma solidity ^0.7.0;

import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "DEXstorage.sol";

contract TestDEXstorage {
   DEXstorage public dexStore;

 address public tok1;
 address public tok2;
 address public address1;
 address public address2;
 bytes4 symbTok1;
 bytes4 symbTok2;

// Run before every test run
 function beforeAll() public {
 tok1 = 0xdf9c2dD7397F7f010E20e2c3b600372c8866Beb4;
 tok2 = 0x11261dB2A31C1b02ac1D8Dbb083c917123De160F;
 symbTok1 = "TOKA";
 symbTok2 = "TOKB";
 address1 = 0x9ef914c91980995913e8b846fbFD25814708bc8A;
 address2 = 0xac0C4bBd7481e6299082a5c1CD2D2e1Bf6291De2;
 }

// Run before every test function
 function beforeEach() public {
  dexStore = new DEXstorage();
  dexStore.addToken(symbTok1, tok1);
  dexStore.addToken(symbTok2, tok2);
 }

// Test that it adds tokens correctly
 function testAddToken() public {
  Assert.equal(dexStore.getToken(symbTok1),tok1,"Address of Token A not correct");
  Assert.equal(dexStore.getToken(symbTok2),tok2,"Address of Token B not correct");
 }

// Test that it adds an offer correctly
 function testAddOffer() public {
  dexStore.addOffer(address1,symbTok1,100,symbTok2,200);
  DEXstorage.Offer offer;
  offer = dexStore.offers[address1];
  Assert.equal(offer.tokenToSell,symbTok1, "Token to sell not correct");
  Assert.equal(offer.tokenToBuy,symbTok2, "Token to buy not correct");
  Assert.equal(offer.amountToSell,100, "Amount to sell not correct");
  Assert.equal(offer.amountToBuy,200, "AMount to buy not correct");
 }
...
}
```

The automated tests follow the standard pattern of resetting the test environment and building the “test fixture” – that is the data needed for testing – before each test call. This is accomplished by the functions “beforeAll()" and “beforeEach()". In this way, the test results do not depend on test ordering. The tests are run by creating a “TestDEXstorage” contract, and sending the proper test messages to it, which is automated by Truffle.

5.6. Design and coding of App System

This subsection covers steps 7 and 8 of ABCDE process. The App System is composed of the software able to present the current 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.

6. Threats to validity

In software engineering, there are three main types of validity that contribute to the overall validity of a research, i.e., internal, construct,

and external validity [58]. Our research regards the proposal of a new software development method for DApps, so its validity assessment is quite different from that of empirical researches.

Internal validity concerns causal relationship between independent and dependent variables, and if there is only one explanation for the research results. In our case, we do not have empirical results on the validity of the method because the application field is relatively new, ABCDE is the first proposed structured method to develop DApps, and some teams are just starting to use it. For these reasons, we believe that internal validity assessment is out of the scope of this threat analysis.

Construct validity concerns the correspondence between the research and the theory that underlies the research itself. From this point of view, a research is valid if one may exclude alternative explanations of the results. In our case, a threat to construct validity might regard the possibility that other approaches might be better than ABCDE for DApp development. For instance, one might argue that a waterfall process combined to a secure software development methodology might better address security concerns of smart contracts. Our experiences with agile methods, which began in the late 90s, and on DApp development make us pretty confident that ABCDE is well balanced between the need to proceed quickly in the presence of uncertain requirements – as it is almost always the case for DApps – without compromising security. Moreover, as reported in section 5, a survey among 14 early developers who used ABCDE gave favorable results. Though, as always in software engineering methods, it is possible to assume that the method assumptions and steps actually work. As always, improvements might be made to ABCDE method, and the release of better methods cannot be ruled out. In this case, ABCDE will still be a yardstick for comparison for other DApp development methods.

Threats to external validity are related to generalisation of our approach: ABCDE was developed specifically for Ethereum DApps. Is it equally valid for DApps intended to run on Ethereum main-net, and on Ethereum permissioned blockchains? How about the applicability to languages different from Solidity, and to other blockchains? Regarding the first issue, we developed ABCDE taking into account both kinds of DApps – for public and permissioned blockchains. The former have typically much stricter security and gas consumption requirements, whereas the latter tend to make use of more complex smart contracts. We balanced the approach considering both issues, addressing security and gas optimization with specific steps, and contract complexity with design diagrams and iterative and incremental development. Therefore, we believe that the validity threat of ABCDE being poorly suited to either public or private blockchain DApps is overcome.

The applicability to other languages and blockchains is a bigger threat, which will be mitigated only by extending ABCDE to cover these subjects. This is work in progress, as outlined in the next section.

7. Conclusions and future work

Despite the substantial inflow of money and the strong efforts made in the development of blockchain-based applications, the application of sound software engineering processes and practices is still quite low. Moreover, DApps development has peculiar characteristics that must be addressed with specific tools and guidelines, and research on these issues has just started, being this field still in its infancy.

We are sure that applying a sound software engineering approach might greatly help to overcome many of the issues of DApp development. These include specific architectural design issues, security issues related to how a blockchain works, the need to spare gas, testing plans and strategies in the blockchain environment, corrective and evolutionary maintenance issues, also related to blockchain immutability.

To our knowledge, the work presented in this paper is the first attempt to develop an organic process for DApp development named ABCDE, from requirement gathering to design, coding, security assurance, and deployment. The proposed method is presently focused on the Ethereum blockchain and its Solidity language, which are at the current time the most used to develop DApps. However, it can be adapted to other environments. For instance, we applied it to model a DApp developed using Hyperledger Fabric [56].

ABCDE takes advantage of agile practices, because DApp development usually deals with rapid implementation of systems whose requirements are not fully understood at the beginning, and tend to change over time. However, given the specificity of blockchains, ABCDE complements the incremental and iterative development through boxed iterations, typical of agility, with more formal tools. These tools include a full modeling of interactions among traditional software and blockchain environment, using UML class diagrams, UML Use Case diagrams (in fact, representing user stories), UML sequence diagrams – all specialized for blockchain-based application development using stereotypes.

ABCDE also provides valuable practices, patterns and checklists to promote and evaluate the security of a DApp written in Solidity language, and also to reduce its gas consumption.

In this paper, we also present an example of application of ABCDE for the development of a simplified Distributed Exchange system to enable trading between pairs of Ethereum tokens. This example is a useful step by step tutorial on the application of guidelines and patterns discussed in the paper.

We believe that ABCDE method can be really valuable to blockchain firms and ICO startups, which might develop a competitive advantage using it since the beginning of their development projects.

Future work will address the improvement suggestions gathered by the survey reported in section 5.1, which include: (i) extending the method to other DApp development environments, such as Hyperledger Fabric; (ii) adding best practices/guidelines for DApp maintenance; (iii) being more specific on the App System development, and on the integration and testing of SC and App systems; (iv) providing teaching materials and more practical examples. We also plan to develop tools such as automated compilers of ABCDE class diagrams into smart contract data structure.

Acknowledgments

This work was partially funded by the CRYPTOVOTING project, funded by Sardinia Region, call POR FESR Sardegna 2014–2020, Prot. 0010083, n. 1361 REA, August 01, 2018, and by the ABATA project (Application of Blockchain to Authenticity and Traceability of Aliments), funded by Italian Ministry for Economic Development, National Operational Program “Enterprises and Competitiveness”, project Nr, F/200130/01–02/X45.

References

[1] S. Nakamoto, Bitcoin: a peer-to-peer electronic cash system, url=, https://bitcoin.org/bitcoin.pdf, 2008.
[2] G. Wood, Ethereum: a secure decentralised generalised transaction ledger, url=, https://ethereum.github.io/yellowpaper/paper.pdf, 2014.
[3] N. Szabo, Smart contracts: formalizing and securing relationships on public networks, First Monday 2, url=, https://ojphi.org/ojs/index.php/fm/article/view/548.
[4] U. Chohan, The Problems of Cryptocurrency Thefts and Exchange Shutdowns, Tech. rep., Discussion Paper Series: Notes on the 21 St Century, School of Business and Economics, University of New South Wales, Canberra, 2018.
[5] 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.
[6] 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, 2018.
[7] 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, 2017, pp. 169–171.
[8] 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.

[9] P. Chakraborty, R. Shahriyar, A. Iqbal, A. Bosu, Understanding the software development practices of blockchain projects: a survey, in: ESEM 2018, October 11–12, 2018, Oulu, Finland, ACM, 2018.

[10] A. Bosu, A. Iqbal, R. Shahriyar, P. Chakraborty, Understanding the motivations, challenges and needs of blockchain software developers: a survey, Empir. Software Eng. 24 (2019) 2636–2673.

[11] K. Schwaber, M. Beedle, Agile Software Development with Scrum, Pearson, 2001.

[12] L. 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.

[13] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, H. Wang, Blockchain challenges and opportunities: a survey, Int. J. Web Grid Serv. 14 (2018) 352–375.

[14] S. Tikhomirov, Ethereum: state of knowledge and research perspectives, in: International Symposium on Foundations and Practice of Security, Springer, 2017, pp. 206–221.

[15] State of the dapps website, url=, https://www.stateofthedapps.com/stats, 2019.

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

[17] 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, 2018, pp. 26–32.

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

[19] D. Janzen, H. Saiedian, Test-driven development concepts, taxonomy, and future direction, Computer 38 (9) (2005) 43–50.

[20] Truffle website, url=, https://www.trufflesuite.com/, 2019.

[21] M. Fowler, Refactoring: Improving the Design of Existing Code, second ed., Addison-Wesley Professional, 2018.

[22] J. Rumbaugh, G. Booch, I. Jacobson, The Unified Modeling Language Reference Manual, Addison Wesley, 2017.

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

[24] 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, 2017, pp. 243–252.

[25] 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, 2018.

[26] 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, 2018.

[27] 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, CEUR Workshop Proceedings, 2019, pp. 43–47.

[28] M. Beller, J. Hejderup, Blockchain-based software engineering, in: Proceedings of the 41th International Conference on Software Engineering Companion, IEEE Press, 2019, pp. 53–56.

[29] V. Lenarduzzi, I. Lunesu, 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, vol. 30, 2018, pp. 1–30, https://doi.org/10.1145/3234152.3234155, 3.

[30] 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.

[31] Y. Huang, Y. Bian, R. Li, L. Zhao, P. Shi, Smart Contract Security: A Software Lifecycle Perspective, IEEE Access, 7.

[32] B. De Win, R. Scandariato, K. Buyens, J. Grégoire, W. Joosen, On the secure software development process: clasp, sdl and touchpoints compared, Inf. Software Technol. 51 (2009) 1152–1171.

[33] K. Rindell, S. Hyrynsalmi, V. Leppänen, Busting a myth: review of agile security engineering methods, in: 12th International Conference on Availability, Reliability and Security, ACM Press, 2017, pp. 1–10.

[34] H. Baumeister, N. Koch, L. Mandel, Towards a uml extension for hypermedia design, in: International Conference on the Unified Modeling Language, Springer, 1999, pp. 614–629.

[35] 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, 2001, p. 10.

[36] 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, 2018.

[37] KPMG, Agile transformation – 2019 survey on agility, url=, https://assets.kpmg/content/dam/kpmg/nl/pdf/2019/advisory/agile-transformation.pdf, 2019.

[38] D.J. Anderson, G. Concas, M.I. Lunesu, M. Marchesi, H. Zhang, A comparative study of scrum and kanban approaches on a real case study using simulation, in: Agile Processes in Software Engineering and Extreme Programming, Springer Berlin Heidelberg, 2012, 23–137.

[39] D.J. Anderson, Kanban: Successful Evolutionary Change for Your Technology Business, Blue Hole Press, 2010.

[40] P. Coad, E. Yourdon, P. Coad, Object-oriented Analysis, vol. 2, Yourdon press, Englewood Cliffs, NJ, 1991.

[41] B. Scriber, A framework for determining blockchain applicability, IEEE Software 35 (2018) 70–77.

[42] L. Marchesi, M. Marchesi, L. Pompianu, R. Tonelli, Security checklists for ethereum smart contract development: patterns and best practices, arXiv preprint arXiv, http://arxiv.org/abs/2008.04761.

[43] L.L. Constantine, L.A. Lockwood, Software for Use: a Practical Guide to the Models and Methods of Usage-Centered Design, Pearson Education, 1999.

[44] H. Sharp, Y. Rogers, J. Preece, Interaction Design: beyond Human-Computer Interaction, fifth ed., John Wiley & Sons, 2019.

[45] Solidity website, url=, https://solidity.readthedocs.io, 2019.

[46] K. Anton, J. Manico, J. Bird, Owasp Proactive Controls for Developers, Tech. rep., Open Web Application Security Project, OWASP), 2018.

[47] Consensys Solidity Best Practices Website, 2019 url=, https://consensys.github.io/smart-contract-best-practices/.

[48] Y. Liu, Q. Lu, X. Xu, L. Zhu, H. Yao, Applying design patterns in smart contracts, in: International Conference on Blockchain, Springer, 2018, pp. 92–106.

[49] 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, 2018, pp. 2–8.

[50] M. Bartoletti, L. Pompianu, An empirical analysis of smart contracts: platforms, applications, and design patterns, in: Financial Cryptography and Data Security. FC 2017. Lecture Notes in Computer Science, Springer, Cham, 2017, pp. 494–509.

[51] Ethereum smart contract security best practices website, url=, https://ethereum-contract-security-techniques-and-tips.readthedocs.io/en/latest/, 2019.

[52] L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino, D. Tigano, Design patterns for gas optimization in ethereum, in: Blockchain Oriented Software Engineering (IWBOSE), 2020 International Workshop on, IEEE, 2020, pp. 9–15.

[53] Proxy patterns, url=, https://blog.openzeppelin.com/proxy-patterns/, 2019.

[54] Solidity patterns, url=, https://fravoll.github.io/solidity-patterns/, 2019.

[55] M. Gupta, Solidity gas optimization tips, url=, https://mudit.blog/solidity-gas-optimization-tips/, 2018.

[56] G. Baralla, A. Pinna, G. Corrias, Ensure traceability in european food supply chain by using a blockchain system, in: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB, 2019, pp. 40–47.

[57] W. Warren, A. Bandeali, 0x: an open protocol for decentralized exchange on the ethereum blockchain, url=, https://0xproject.com/pdfs/0x_white_paper.pdf, 2017.

[58] R. Conradi, A.I. Wang (Eds.), Empirical Methods and Studies in Software Engineering, vol. 2765, LNCS, Springer Berlin Heidelberg, 2003.