On Refining Design Patterns for Smart Contracts

Marco Zecchini[1 [0000−0002−2280−9543]], Andrea Bracciali[2 [0000−0003−1451−9260]], Ioannis Chatzigiannakis[1 [0000−0001−8955−9270]], and Andrea Vitaletti[1 [0000−0003−1074−5068]]

[1] Sapienza University of Rome, Italy
(1,b,c)@diag.uniroma1.it
[2] University of Stirling, UK
abb@cs.stir.ac.uk

Abstract. The need for a Blockchain Oriented Software Engineering (BOSE) has been recognized in several research papers. Design Patterns are considered among the main and compelling areas to be developed in BOSE. Anyway, design patterns need to be enhanced with some additional fields to better support the specific needs of Blockchain development. In this paper, we discuss the use of Solidity design patterns applied to a water management use case and we introduce specific fields in their description, aiming at offering to Blockchain developers more support in the critical decisions to build efficient decentralized applications.

Keywords: Design Patters · Blockchain Oriented Software Engineering (BOSE) · Use Case · Smart Contracts · Solidity

1 Introduction

Since the release of Ethereum, there have been many cases in which the execution of Smart Contracts managing Ether coins has led to problems or conflicts. Probably, the most well known example of such issues is "The DAO" [15, 13]. The DAO, decentralized autonomous organization, was a concrete attempt to implement a funding platform, similar to Kickstarter, running over Ethereum. It went live in 2016 with between 10-20 thousand investors (estimation) providing the equivalent of about US$ 250 million in funding and thus breaking all existing crowdfunding records. However, after few months an unintended behavior of the DAOs code was exploited draining the fund of millions of dollars worth of ETH tokens. The DAO experience makes clear the importance of suitable Blockchain Software Engineering (BOSE) techniques, capable to reduce the risks connected to "poorly" designed and implemented smart contracts. However, a discipline of Smart Contract and Blockchain programming, with standardized best practices is yet in its infancy and requires new approaches since smart contracts rely on a non-standard software life-cycle; as an example, once deployed applications can hardly be updated or bugs resolved by releasing a new version of the software.

In [12] the authors discuss the need for BOSE and propose three main areas for the initial development of this discipline: a) Best practices and development methodology, b) Design patterns and c) Testing.

Our work is focused on the use of Design Patterns for the development of Decentralized Applications (DApps) [25].

DApps are a new class of applications coded in programs running on the blockchain. DApps may provide a variety of services over the underlying P2P infrastructure that up to now have only been provided in the dominant Client/Server architectures. The Peer-to-Peer (P2P) nature of DApps and the lack of a central authority as in the Client/Server paradigm, is the key ingredient to implement infrastructures supporting new forms of democratic engagement of the users. A recent report by Fluence Labs [17] presents the state of the DApps ecosystem surviving 160 projects. The main findings can be summarized in the following points: a) DApps is a modern trend: 72% of the projects started in 2018, b) 87% of the projects run on Ethereum c) A quarter of the surveyed projects are gaming DApps. d) About half of the projects used a centralized tools to connect to the Ethereum blockchain. e) Transactional fees prevailed as the central monetization model for most projects. f) New user onboarding was mentioned by more than three quarters of the respondents as the major obstacle to adoption.

Problem Statement. Design Patterns are undoubtedly a useful tool to improve the development of DApps. Due to the nature of the Blockchain, more than in other contexts DApps are at risk of generating problems, which are hard to recover from. However, most of the available Design Patterns for Blockchain do not consider some useful information that can help the developer to implement correct and efficient solutions.

Contribution of the paper. In this paper we analyse and evaluate best practices in the use of Design Patterns for a typical DApp. Moreover, we refine the format description of design pattern specific for blockchain with other fields, namely *Cost of execution* and *Decentralization level* and *On-chain/Off-chain components*, capable to help developers in trading-off between critical design and implementation choices. As a running example, we develop a DApp interacting with Internet Of Things devices to monitor and manage resource consumption, and encourage the democratic engagement and empowerment of citizens. In particular, our use case focuses on a DApp for the management of urban water resources. However, the Design Patterns employed in this specific use case are of general interest and can support a variety of other application scenarios.

2 Use Case: decentralized management of urban water resources

The interactive statistics portal by the International Water Association (IWA) [8] provides data on water consumption, tariff structure and regulation of water services in 198 cities in 39 countries from all 5 continents. The IWA report stresses the importance of adopting modern emerging technologies and smart metering to improve the overall water management process in the city.

In nowadays cloud solutions, data on water consumption is collected by smart meters and delivered to a cloud service allowing live monitoring of the consumption and providing evidences on consumption patterns to the users in the hope that citizens will consequently act to reduce their water consumption.

It is worth noting that the overall success of such resource management strongly depends on the active and collective participation of citizens. The virtuous behaviour of an individual is commendable, but the risk is that it is literally "a drop in the ocean" if not accompanied by the joint action of the community.

Nowadays, the Client/Server paradigm dominates the cloud services market. However, decentralized applications (DApps), an emerging P2P framework, have the potential to revolutionize this market and democratize the whole process. The hope is that the empowerment of the citizen, guaranteed by this democratic process, will improve the participation and thus the overall chances of success of smart cities initiatives. As observed in [22], smart cities often do not optimally reach their objectives if the citizens are not suitably involved in their design.

We assume the availability of suitable Internet of Things (IoT) devices, i.e. smart water meters, capable to measure the performance of a target process, i.e. reducing water consumption. However, we want then to have citizens engaged in pursuing behavioural changes. It is well-known that behavioural changes are the effective way to better performances in resource management, e.g. closing the water while brushing teeth can save up to 20 litres, and taking a shower can save up to four times the water necessary for a bath.

To support the active engagement and democratic participation of users, our proposed DApp will implement two main principles:

- Citizens propose smart contracts that encode measures of the effectiveness of water management policies. Typically, such smart contracts relies on IoT data to monitor the application of policies. Citizens also select a smart contract by a fair vote. Such smart contracts, capable of attracting the greater consensus from citizens, becomes currently operative. Selection may occur regularly, on demand, or on specific conditions.
- To further encourage the participation of citizens, the operative smart contract will also be in charge of distributing incentives to virtuous citizens, namely citizens that most actively contribute to fulfill policies, i.e. successfully reduce the water consumption.

2.1 Reference Architecture

The full implementation of the Proof of Concept (PoC) is available in the GitHub repository [2] and a simplified reference architecture is shown in figure 1.

Smart Contracts The smart contracts have been developed in Solidity [11], the object-oriented, high-level language for implementing smart contracts on

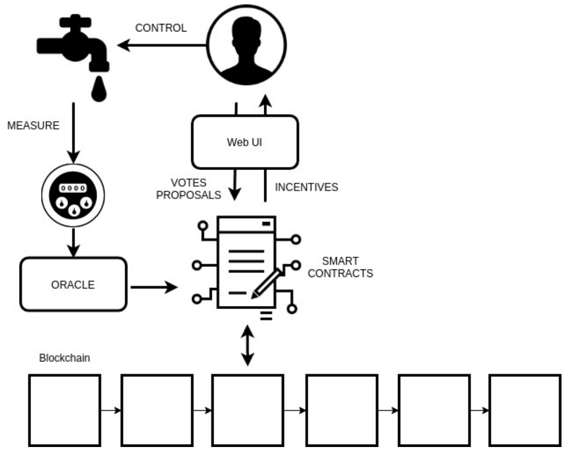


Fig. 1. A simplified picture of a DApp for water management. Contrary to centralized architectures the back-end of this architecture is implemented on smart contracts running on the P2P decentralized blockchain infrastructure.

Ethereum, using Remix [4], the browser-based compiler and IDE that enables users to build Ethereum contracts and to debug transactions.

The *manager* smart contract, implemented in the *ManagerContract* class, is in charge to a) manage the interface with IoT sensors and safety of data, b) manage the voting process at each occurrence of it, and c) make operational the most voted proposal, which will monitor the application of policies and devolve incentives accordingly. Several variations to this general scheme are possible, of course, but for the sake of this paper such a general formulation will be adequate. In the next section we discuss the design patterns involved in the design of the proposed DApps for urban water management. The voting process code is inspired by the example on ballot available on Solidity documentation [5] . The vote starts and finishes, respectively, with the methods *startVote* and *winningProposal*. The latter, takes also care of counting the votes and electing the winner. The *addProposalContract* method allows users to propose a new contract, while the method *vote* allows user to vote for a proposal. Note that this method is *payable* because it has to collect the funding during the voting process and to transfer the accumulated funding to the winning proposal using the method *trasferToWinner*.

The *proposal* smart contract must essentially define a) how to measure the contribution by each citizen to the reduction of water consumption as measured by IoT sensors, and b) how to distribute incentives according to that contribution. Citizens are free to present proposals, namely alternative solutions that are democratically voted by the citizens themselves. The class *ProposalContract* represents a proposal of a citizen. The *owner* variable maintains the owner of the contract (this is crucial for the implementation of the access restriction pattern, see section 3). As already observed, the ManagerContract manages all the key phases of the process and consequently, before starting the voting phase, every participating contract should change its ownership to the ManagerContract calling *changeOwner* method.

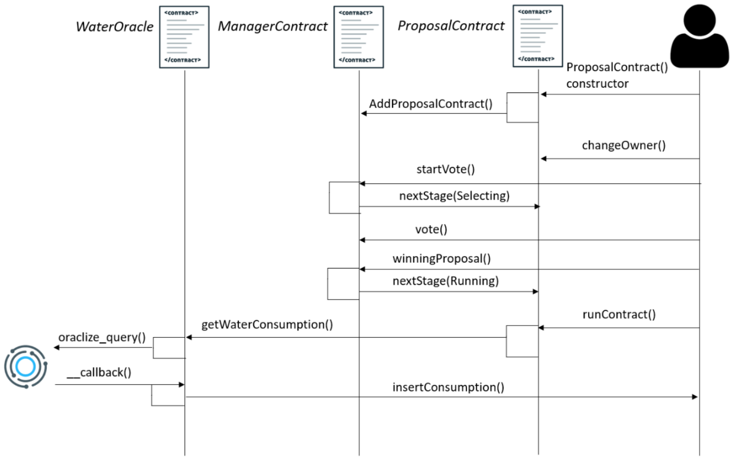


Fig. 2. Interaction with the Smart Contracts

Finally, the proposal that wins the election phase become runnable and the corresponding smart contract is executed invoking the method *runContract*. This results in a call to the smart contract in charge of collecting the data on water consumption interacting with the oracle. This data are stored in the Proposal-Contract by the *insertConsumption* method. Each citizen is allowed to access only its own consumption data. The hash table *consumers* maintains the association between the user credentials that are used by the oracle to access the data, and their address in the blockchain.

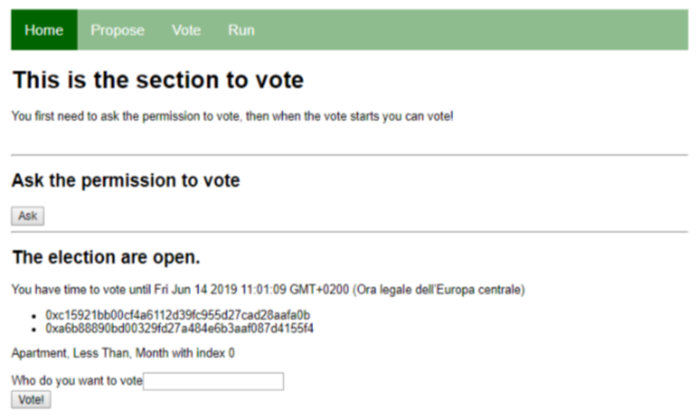
WaterOracle is the smart contract that collects the data on the water meters from a source of data external to the blockchain.

The whole process, summarized in figure 2, which allows the citizens to select the smart contract that will become operative can be divided in three phases: 1) the proposal phase, 2) the selection phase and 3) the running phase. During the proposal phase, proposal contracts are submitted by the community. In the selection phase, proposal contracts are voted by the community. For the sake of simplicity, we assume that citizen can access the vote only by providing a fee that is used to accumulate the incentives. More realistic fee policies are scope for future work. The voted contract becomes operative in the running phase, and will actually distribute the accumulated incentives according to its own policies.

Web UI. Users can access the functions of the system by *a web application* running on his/her premises that interacts with the blockchain. This app is developed in Nodejs and web3.js[6], which is a collection of libraries allowing to interact with a local or remote Ethereum node, through HTTP, WebSocket and IPC connection. While in principle citizens can make their proposals in the form of free smart contracts encoded in solidity, in this paper we focus on a template smart contract where the proposals are characterized by different parameters that users can freely select accessing the *Proposal* page of the Web UI. Examples of the parameters defining the smart contracts are: *What* will be monitored (e.g. apartment, building), the *Criteria* to distribute the incentive (e.g. $\leq$ a given threshold) and the *Interval* in which the monitoring activity is performed and incentives are distributed (e.g. semester).

Once a proposal has been formalized, it can be voted accessing the *Vote* page, showed in figure 3, where users can inspect the proposals and finally vote for the most liked one. After a suitable time interval, that allows each voter to express their preferences, the winner proposal is elected and starts the running phase.

In the *Run* page the user can finally allow the selected smart contract to access its data on water consumption. If the criteria defining a virtuous behaviour embodied in the smart contract are meet, the corresponding incentives are automatically sent to the user.



This is the section to vote

You first need to ask the permission to vote, then when the vote starts you can vote!

Ask the permission to vote

The election are open.

You have time to vote until Fri Jun 14 2019 11:01:09 GMT+0200 (Ora legale dell'Europa centrale)

- 0xc15921bb00cf4a6112d39fc955d27cad28aafa0b
- 0xa6b88890bd00329fd27a484e6b3aaf087d4155f4

Apartment, Less Than, Month with index 0

Who do you want to vote

Fig. 3. The Voting Interface

3 Smart Contract Design Patterns

We used some of the Solidity design and programming patterns [24, 9, 25] collected by Franz Volland in his github repository [23]. The aim of this section is double: to discuss how these design patterns have been employed to implement the methods and the smart contracts in the proposed decentralized system for water management; to describe how a design pattern is expanded in the case of a blockchain design pattern (BDP).

The documentation for a design pattern describes the context in which the pattern is used, the forces within the context that the pattern seeks to resolve, and the suggested solution. A variety of different formats [14] have been used by different pattern authors. [23] uses one of these approaches. We intend to add two additional fields for describing a BDP:

– The first is *cost of execution - gas*, i.e. the unit to measure the amount of computational effort to execute certain operations. Its presence is fundamental and necessary for public blockchains such as Ethereum: this in fact avoids that an operation performs forever on the blockchain blocking the entire network.
– Secondly, there are the *blockchain specific features* which are a set of properties that highlights how BDP are related with peculiar characteristics of blockchains. We have identified decentralization and on-chain or off-chain properties.

For the sake of space we don't sketch the code of all patterns, the interested reader is referred to [2] and we do not report the formal description of the patterns already available at [23], but we focus our attention to the two additional descriptive fields presented above.

Ownership and Access Restriction pattern During the proposal phase, users make contract proposals. In order to participate to the next selection phase, users have to release contract ownership to the manager smart contract. This is done by implementing the *Access Restriction* pattern which allows the ownership of a contract to be changed. The proposer invokes the *changeOwner* function , providing as input the address of the manager that consequently becomes the owner. We stress here that at each instant in time there is only one owner for a contract and some functions can be invoked only by the owner because they are critical for the correct execution of the contract.

Cost of execution - gas.	
changeOwner()	28595
Blockchain specific features	
Decentralization of the BDP	Decentralized BDP
On-chain or off-chain solution.	The owner is stored into a variable of the smart contract, so it is an on-chain solution.

State Machine pattern In each phase, a proposal contract can be in one of three possible states: proposal, selection and running. Only the owner of a contract can change the status of the contract.

The *State Machine* pattern [23] allows a contract to go through different states, with different functions enabled in the different states. A function modifier checks if the contract stage is equal to the required stage before executing the called function. Note that the manager, becoming the owner of the contracts, is the only one capable to change the state of a contract during the selection phase (see Access Restriction pattern: onlyBy(owner)).

Cost of execution - gas.	
nextStage()	27632
Blockchain specific features	
Decentralization of the BDP	Decentralized BDP
On-chain or off-chain solution.	It changes the internal state of a smart contract that lives on-chain

Oracle pattern Once in a running state, the winning smart contract needs to collect data from the smart meters to correctly dispense incentives to the users. This requires the communication with an Oracle, a centralization point, to gain access to data outside the blockchain. An Oracle is hence a trusted entity providing a unique view on a source of data considered credible.

Each node in the blockchain has to validate every computation performed in a smart contract. When this requires the interaction with off-chain sources of data, as in our case with smart meters, this becomes unpractical because, due to network issues (e.g. delays), there are not guarantee that all the node will access the same information as expected thus leading to a possible break in the consensus algorithm.

In our PoC, we use the oracle service provided by Oraclize [3], see listing 1 (recently Oraclize changed its name to Provable).

Listing 1. The call of an Oracle to acquire the water meter readings and send them back to ProposalContract.

```
contract WaterOracle is usingOraclize {
    uint public water;

    function () public  payable {}
    function getWaterConsumption(string input_for_API)
    public {

        if (oraclize_getPrice("URL") > this.balance) {
            emit LogError("Put more ETH");
        }
        else {
            //call the oracle and save the request
        }
    }
    function __callback(bytes32 myid,
        string _result) public
    {
        //update consumption

    }
}
```

The function *getWaterConsumption* is invoked by the ProposalContract and performs the query to the oracle. The fallback function is necessary to support the necessary payments to the Oracle: only if the balance of the WaterOracle smart contract is sufficient, the query is delivered to the Oraclize contract that access the data interacting with the data source API. Once data are available a *__callback* function is called to store the values on the ProposalContract in the public variable *water*. The value of *water* is finally used to distribute the incentives.

Cost of execution - gas.	
WaterOracle deployment	1362206
getWaterConsumtion()	144520
Blockchain specific features	
Decentralization of the BDP	Most oracles are points of centralization within the network. However projects on decentralized Oracles exists, such as ChainLink [1] which Provable, the new brand behind Oraclize, now supports.
On-chain or off-chain solution.	This pattern can be implemented either partly on-chain and off-chain (an oracle smart contract with external state injected by an off-chain injector) or totally off-chain (external server signing transactions). [25]

3.1 Discussion

Design Patterns are descriptions or templates to solve problems that can emerge in many different situations, and consequently are usually not a finished design that can be transformed directly into code [14]. However, in the Blockchain, the implementation details have direct consequences on the execution costs of a given pattern that are crucial to determine the feasibility and the success of a project. If the costs of running a system are higher than the expected benefits, users will possibly not participate in the initiative.

As far as concerns the level of decentralization this is crucial to support the democratization of an initiative and thus the active participation of the users, but can have a cost. Let's consider the oracle example. The simplest solution that relies on a single "centralised" oracle is likely the most cost effective. We can reduce the centralization requiring the same information to n independent oracles, but even assuming that we can get the exact same information (e.f. time and source) from all of them, this will result in a cost n-times higher.

The introduction of quantitative metrics (i.e. gas) to evaluate design patterns is not novel (see [7] and [10]) and necessarily require the implementation of the considered design patterns.

4 Related work

The need for a blockchain-oriented software engineering (BOSE) is recognised in [19] where the authors suggest that ensuring effective testing activities, enhancing collaboration in large teams, and facilitating the development of smart contracts all appear as key factors in the future of blockchain-oriented software development. Compared to traditional Software Engineering, BOSE is not yet well developed and Smart Contracts rely on a non-standard software life-cycle. As an example, once delployed, they can be hardly updated and even simple bugs are difficult to fix. [12] suggests to focus on three main areas for the development of BOSE: a) Best practices and development methodology, b) Design patterns and c) Testing.

In [9] the authors quantify the usage of smart contracts on Bitcoin and Ethereum in relation to their application domain and analyse the most common programming patterns in Ethereum.

Due to the inherent nature of blockchain based contract execution, missing low level programming abstractions, and the constant evolution of platform features and security considerations, writing correct and secure smart contracts for Ethereum is a difficult task. In [24] the authors mined a number of design patterns providing design guidelines and showed that those patterns are widely used to address application requirements and common problems.

The literature on blockchain technologies in the smart cities has been recently reviewed in [21]. The paper analyses a number of sectors where the blockchain can contribute to build a smarter city, including water management. A privacy-friendly blockchain-based gaming platform aiming at engaging users in reducing water or energy consumption at their premises is proposed in [20], but this paper does not explicitly use smart contracts.

In [18] the authors stress that lack of transparency and trust on a centralized network infrastructure could be a key factor that hinders the true realization of the citizen participatory governance model. Our proposed DApp is an example of smart urban collaboration implemented over a P2P network thus overcoming most of the limits of traditional centralized networks and guaranteeing an unprecedented level of transparency and trust. In the blockchain, the trust shift from a single and centralized third party to the whole P2P infrastructure, that is decentralized in its nature.

Voting is considered among the most important application of the blockchain technology in the public sector [16]. In our proposed approach, voting is used to select which among the proposed contracts will become actually operative. A fully aware vote requires the understanding of smart contracts and their implications and we cannot expect this is within everyone's reach. The research on the methods to wider the audience capable of understanding smart contracts is out of the scope of this paper. In our implementation, we propose a smart contract template where users can simply and freely select some of the key parameters defining the contract.

.

5 Conclusion

In this paper we discussed the applicability of solidity design patterns to the development of decentralized application (DApp) for urban water management. The decentralized nature of DApp implements a democratic process that will hopefully encourage the active participation of the citizen to the actions necessary to reduce the water consumption. Design patterns are among the key ingredients that have been identified to develop a blockchain-oriented software engineering (BOSE) capable to reduce the risks connected to the unique life-cycle of smart contracts. The main contribution of the paper can be summarized in the following points:

- Moving from a centralized Client/Server architecture, typical of current implementations of smart city service, to DApps will remove the necessity of trusting central authorities, which is considered one of the most relevant factors that limit the true realization of citizen participatory governance [18].
- The code of the proposed DApp is available on the github repository [2].
- We propose an extension of the design patterns considering two additional fields, namely *cost of transaction* and *blockchain specific feature* that helps developers in implementing a more effective DApp.
- The proposed extension has been discussed in the implementation of the three design patterns [23] employed in the proposed DApp.

References

1. Chainlink web site. `https://https://chain.link/`, 2019. [Online; accessed May-2019].
2. Dapp_water. `https://github.com/marcozecchini/Dapp_Water`, 2019. [Online; accessed June-2019].
3. The provabletm blockchain oracle for modern dapps. `https://provable.xyz/`, 2019. [Online; accessed May-2019].
4. Remix. `https://remix.ethereum.org`, 2019. [Online; accessed May-2019].
5. Solidity documentation. `https://solidity.readthedocs.io/`, 2019. [Online; accessed May-2019].
6. web3.js - ethereum javascript api. `https://web3js.readthedocs.io/en/1.0/`, 2019. [Online; accessed June-2019].
7. Apostolos Ampatzoglou and Alexander Chatzigeorgiou. Evaluation of object-oriented design patterns in game development. *Information and Software Technology*, 49(5):445–454, 2007.
8. International Water Association. Water statistics. `http://waterstatistics.iwa-network.org/`, 2019. [Online; accessed May-2019].
9. Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security*, pages 494–509, Cham, 2017. Springer International Publishing.

10. Angelo Corsaro and Corrado Santoro. The analysis and evaluation of design patterns for distributed real-time java software. In *2005 IEEE Conference on Emerging Technologies and Factory Automation*, volume 1, pages 8–pp. IEEE, 2005.
11. Chris Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, Berkely, CA, USA, 1st edition, 2017.
12. G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons. Smart contracts vulnerabilities: a call for blockchain software engineering? In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 19–25, March 2018.
13. Q. DuPont. Experiments in algorithmic governance a history and ethnography of “the dao”, a failed decentralized autonomous organization. In M. Campbell-Verduyn, editor, *Bitcoin and Beyond: Cryptocurrencies, Blockchains, and Global Governance*, chapter 8, pages 157–176. Routledge, 2017.
14. Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.
15. Mehar M. I., Shier C. L., Giambattista A., Gong E., Fletcher G., Sanayhie R., Kim H. M., and M. Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain: The dao attack. *Journal of Cases on Information Technology (JCIT)*, 21(1):19–32, 2019.
16. Nir Kshetri and Jeffrey Voas. Blockchain-enabled e-voting. *IEEE Software*, 35:95–99, 07 2018.
17. Fluence Labs. Dapp survey results 2019. `https://medium.com/fluence-network/dapp-survey-results-2019-a04373db6452`, 2019. [Online; accessed May-2019].
18. Albert Meijer and Manuel Pedro Rodrguez Bolvar. Governing the smart city: a review of the literature on smart urban governance. *International Review of Administrative Sciences*, 82(2):392–408, 2016.
19. S. Porru, A. Pinna, M. Marchesi, and R. Tonelli. Blockchain-oriented software engineering: Challenges and new directions. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 169–171, May 2017.
20. C. Rottondi and G. Verticale. A privacy-friendly gaming framework in smart electricity and water grids. *IEEE Access*, 5:14221–14233, 2017.
21. C. Shen and F. Pena-Mora. Blockchain for citiesa systematic literature review. *IEEE Access*, 6:76787–76819, 2018.
22. A. Simonofski, E. S. Asensio, J. D. Smedt, and M. Snoeck. Citizen participation in smart cities: Evaluation framework proposal. In *2017 IEEE 19th Conference on Business Informatics (CBI)*, volume 01, pages 227–236, July 2017.
23. Franz Volland. Solidity patterns. `https://fravoll.github.io/solidity-patterns/`, 2019. [Online; accessed May-2019].
24. Maximilian Wohrer and Uwe Zdun. Design patterns for smart contracts in the ethereum ecosystem. *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1513–1520, 2018.
25. Xiwei Xu, Ingo Weber, and Mark Staples. *Architecture for Blockchain Applications*. 03 2019.