# Applying Design Patterns in Smart Contracts

**Chapter** · June 2018

DOI: 10.1007/978-3-319-94478-4_7



**5 authors**, including:

Qinghua Lu
The Commonwealth Scientific and Industrial Research Organisation
**87** PUBLICATIONS **537** CITATIONS
SEE PROFILE

Xiwei Xu
The Commonwealth Scientific and Industrial Research Organisation
**83** PUBLICATIONS **1,145** CITATIONS
SEE PROFILE

Liming Zhu
The Commonwealth Scientific and Industrial Research Organisation
**178** PUBLICATIONS **2,623** CITATIONS
SEE PROFILE

# Applying Design Patterns in Smart Contracts
## A Case Study on a Blockchain-Based Traceability Application

Yue Liu[1], Qinghua Lu[1,2,3(✉)], Xiwei Xu[2,3], Liming Zhu[2,3], and Haonan Yao[1]

[1] College of Computer and Communication Engineering, China University of Petroleum (East China), Qingdao, China
qinghua.lu@data61.csiro.au
[2] Data61, CSIRO, Sydney, Australia
[3] School of Computer Science and Engineering, UNSW, Sydney, Australia

**Abstract.** Blockchain, the technology Bitcoin lives on, is an emerging research field due to its nature of decentralisation, and properties of data immutability and transparency. Smart contracts are the programs executed on programmable infrastructure provided by blockchain, which can manage complex business logic, extending the field significantly. As blockchain technology is still at an early stage, there are little works on applying software architectural methods to the design of blockchain-based applications. In this paper, we summarise eight smart contract design patterns based on existing smart contracts and our experience, and classify them into four categories: *Creational Patterns*, *Structural Patterns*, *Inter-Behavioral Patterns*, and *Intra-Behavioral Patterns*. We share some experiences of applying the presented design patterns of smart contract on a real-world blockchain-based traceability application, and also discuss how patterns can improve the quality attributes of blockchain-based application.

**Keywords:** Blockchain · Smart contract · Interoperability
Adaptability

## 1 Introduction

Blockchain, the technology behind Bitcoin [5], is a decentralised append-only data store, where all participants in the network can reach agreements on the states of transactional data, without relying on a centralised system. Data transparency and immutability are the key characteristics of blockchain technology, which can help prevent tempering or revising the submitted transactions on blockchain.

Besides the distributed ledger as a data storage, blockchain provides a general-purpose programmable infrastructure. *Smart contracts* [6] are programs deployed and running on blockchain, which can express triggers, conditions and


S. Chen et al. (Eds.): ICBC 2018, LNCS 10974, pp. 92–106, 2018.
https://doi.org/10.1007/978-3-319-94478-4_7

business logic [9] to enable more complex programmable transactions. Many startups, enterprises, and governments [7] are currently exploring blockchain applications in areas as diverse as supply chain, electronic health records, voting, energy supply, ownership management, identity management, and protecting critical civil infrastructure. However, since blockchain technology is still at an early stage, there are little works on applying software architectural methods to the design of blockchain-based applications, particularly design of smart contracts.

In software architecture community, a blockchain taxonomy has been proposed to compare different blockchain platforms and assist in the design and evaluation of software architectures using blockchain technology [11]. Other than taxonomy, architectural design patterns is also a mechanism to classify and organise the existing solutions.

A design pattern is a reusable solution to a problem that commonly occurs within a given context during software design [3]. We investigate some existing patterns for distributed system, peer-to-peer system and software design patterns in general, and assess the applicability of the existing patterns to the design of smart contracts. The study results in the experiences that there are some reusable solutions that can be applied to the design of smart contract in a blockchain-based system.

In this paper, we first summarise and classify eight smart contract design patterns. The patterns are divided into four categories: *Creational Patterns*, *Structural Patterns*, *Inter-Behavioral Patterns* and *Intra-Behavioral Patterns*. By using the patterns, blockchain can not only be used for storing or exchanging data, but also handle with more complicated programs with complex logic, which can benefit developers on building blockchain-based applications. Besides, we use a real-world blockchain-based traceability system, originChain, as a case study to show how to apply design patterns to smart contracts. The architecture design of originChain is briefly discussed in our previous work [4]. This paper focuses more on the structural design of smart contracts, gives more details of several design patterns, and we also share some experiences of applying the patterns to improve the quality attributes of originChain, such as adaptability and interoperability.

The remainder of this paper is organised as follows. Section 2 discusses background information and introduces the related work. Section 3 summarises design patterns for smart contracts and classifies them into four categories. Section 4 presents how to apply those patterns on a blockchain-based traceability application. Section 5 discusses the lessons learned from this case study. Section 6 concludes the paper and outlines future work.

## 2 Background and Related Work

### 2.1 Blockchain and Smart Contracts

When Bitcoin was released to public, its capability was limited, providing merely a public ledger to record the transactions related to a specific digital cryptocurrency [8]. Since a programmable infrastructure called *smart contract* has been

deployed, the blockchain technology is considered enhanced, as it is enable to deal with more complex transactions, such as triggers, conditions and business logic [9], what users need to do is to authorise their operations via cryptographic signature.

*Solidity*, a Turing-complete programming language for writing smart contract, is supported by several blockchain platforms that implement Ethereum Virtual Machine[1] such as Ethereum and Parity. *Solidity* is similar to the object-oriented programming languages, a contract in *Solidity* can be considered as a "class" in Java. In addition, *Solidity* also has the mechanisms such as interface, inheritance, and exception, etc. Due to such properties of *Solidity*, it is possible to apply existing design patterns to the *Solidity*-based smart contracts.

### 2.2 Designing Blockchain-Based Applications

Although blockchain is young, there are a lot of enterprises, institutions, and governments all over the world who are interested in this technology and investigating the applications based on it. Big companies, like Microsoft[2], IBM[3], Amazon[4] provide convenient and instant service of building up a private blockchain. Such works are considered as the combination of cloud service and blockchain technology.

In academia, from the perspective of software architecture, blockchain technology has been considered as a connector to store data in software architecture [10], and the trade-off analysis of choosing blockchain instead of traditional centralised database was discussed in [11]. P. Zhang and his colleagues [12], shared the experience of designing a blockchain-based healthcare platform, to which they applied several software patterns to improve the application scalability.

There are some works on design patterns of smart contract for blockchain-based application. In [2], J. Eberhardt and S. Tai proposed four patterns, including challenge response pattern, off-chain signatures pattern, content-addressable storage pattern, delegated computation pattern, and low contract footprint pattern, which mainly focus on the separation of on-chain and off-chain for data and computation. Bartoletti and Pompianu [1] demonstrated an empirical analysis of smart contracts, in which they collected hundreds of smart contracts and divided them into several categories: token, authorisation, oracle, randomness, poll, time constraint, termination, math and fork check. However, each kind of the smart contracts presented in this analysis has a specifically functional feature, there is a lack of a systematic analysis on smart contracts against architectural properties, which is the focus of this paper. This study focuses more on the architecture design of smart contracts, we emphasise the interoperation among contracts, and present the pseudocode of some patterns.

[1] https://solidity.readthedocs.io/en/develop/.
[2] https://azure.microsoft.com/en-us/solutions/blockchain/.
[3] https://www.ibm.com/blockchain/.
[4] https://amazonaws-china.com/cn/partners/blockchain/.

## 3 Design Patterns of Smart Contract

In this section, we summarise several design patterns [3], and discuss the technical details of each pattern. The patterns are categorised as following: (1) *Creational Pattern* abstracts the process of instantiation, helping developers create smart contracts independently, (2) *Structural Pattern* focuses on the relationship between contracts and their instances, (3) *Inter-Behavioral Pattern* can enhance the flexibility when contract instances need to operate with each other, (4) *Intra-Behavioral pattern* is aimed to improve a particular property, like interoperability or adaptability of the whole system.

### 3.1 Interaction Among Design Patterns

Figure 1 illustrates a high-level design of applying eight software patterns in a blockchain-based application. Developers can instantiate a contract instance through *Contract Composer*, if the instance contains several objects, and *Contract Factory*, if the instance has a relatively simple structure, and the address of instance is stored in a *Contract Facade* instance for optimal management. Before a contract instance in *Contract Facade* is called, the authenticity of the caller needs to be examined by *Hash Secret* or *Multi-Signature*, depending upon the situation. The caller's identification result is checked by server, if it is valid, the operations will be passed to *Contract Mediator*, which carries out the implementation. When there are new requirements according to the issue of regulation in a certain industry area, *Contract Decorator* helps update a particular contract instance or the whole *Contract Composer* into a new one, by encapsulating the old-version via contract address and appending the new requirements. The new-version contract instances have different addresses, thus *Contract Observer* is needed to update the corresponding contract information in *Contract Facade*.

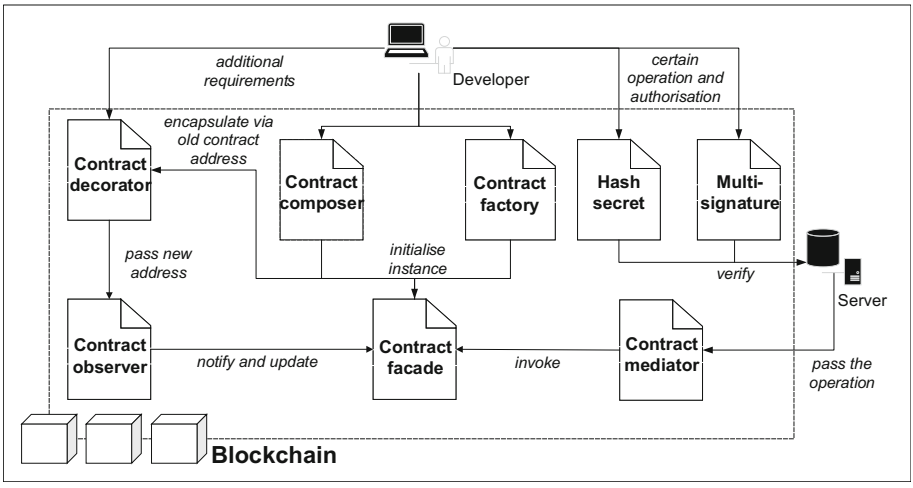


**Fig. 1.** Structural design of smart contracts.

### 3.2 Creational Pattern

**Contract Factory.** As the compiled code of a smart contract deployed on blockchain is not readable, it is tedious to deploy and manage smart contracts that have same properties but aim to diverse clients. With the help of this pattern, developers do not need to deploy the smart contracts one after another, but deploy a contract factory once, through which the required multiple instances can be instantiated.

**Contract Composer.** In a blockchain-based application, the combination of services or objects is inevitable. Consequently, how to effectively control such a combination becomes a challenge to developers, especially under the condition that each service or object is represented in the form of smart contract. Compared with *Contract Factory*, *Contract Composer* focuses on the complex structure of a contract instance, as it can construct a complicated target through multiple small pieces.

### 3.3 Structural Pattern

**Contract Decorator.** Once a smart contract is deployed on blockchain, it is not allowed to modify or update the source code of that contract. *Contract Decorator* pattern can avoid rewriting the whole contract when there are new requirements, developers just need to encapsulate the old contracts and append the required features into a new version of the contract through this pattern, to achieve updatability and modifiability.

**Contract Facade.** Managing smart contracts may be a burdensome work as there are massive contracts having similar features in a blockchain-based system. *Contract Facade* pattern can relieve such pressure via providing a simple interface by coping with contract addresses. Such an interface is also in the form of smart contract, for developers to call the functions of similar contracts.

### 3.4 Inter-behavioral Pattern

**Contract Mediator.** In a business process, smart contracts need to interact with each other to finish a certain activity, which may result in tight coupling of the contracts. *Contract Mediator* pattern aims to reduce the communication complexity of smart contracts, an instance of this pattern is in the form of smart contract, which collects and encapsulates the interactions and invocations from one contract to the others, to decoupling the smart contracts.

**Contract Observer.** When a smart contract is modified due to the changing requirements in industry, all the related contracts need to be informed and updated automatically. *Contract Observer* pattern can deal with such problem

to achieve interoperability and updatability among the contracts via an observer instance. An instance of *Contract Observer* needs to define the objects and information involved, once there are any changes, it should notify all the objects to update information.

### 3.5 Intra-behavioral Pattern

*Intra-Behavioral Patterns* do not contribute to the contract architecture as much as the three categories mentioned above, but each one has the ability to work both independently and collaboratively. In this study, *Hash Secret* and *Multi-Signature* are proposed, they have at least one specific property to advance the non-functional requirements of a blockchain-based application respectively.

**Hash Secret.** This pattern can help a user to achieve authorisation of a particular activity to unknown authorities, by generating a digital secret key known as the hash secret. When the authority is decided, it will then receive the hash secret and thus have the ability to finish further task.

**Multi-signature.** As there are multiple authorities in a blockchain network, this pattern can provide a flexible way to achieve better cooperation. A transaction is valid only when there are enough signatures from the authorities. In addition, this pattern can also be considered as an individual safeguard mechanism as the current blockchain technology does not provide a way to recover the lost private key.

## 4 Applying Design Patterns in the Traceability System

In this section, we apply five of the above patterns to a real-world blockchain-based traceability system proposed in [4]: *Contract Composer*, *Contract Facade*, *Contract Observer*, *Hash Secret*, and *Multi-Signature*. *Contract Factory* is commonly used in many blockchain-based applications, thus in this paper we decide not to demonstrate this pattern. *Contract Mediator* and *Contract Decorator* need further refinement, consequently these two patterns will be included in our future work.

### 4.1 Traceability Systems

Tracking products during production and distribution for the relevant product information (e.g., originality, transportation route and location, and quality certificate) is the main goal of a traceability system. Figure 2 demonstrates the business process of product traceability. A government-certified traceability company can provide traceability services to product suppliers and retailers, by sending staffs to inspect major operation flow, namely to examine factory and freight yard, and contact third-party labs to do sample testing, and if the requirements

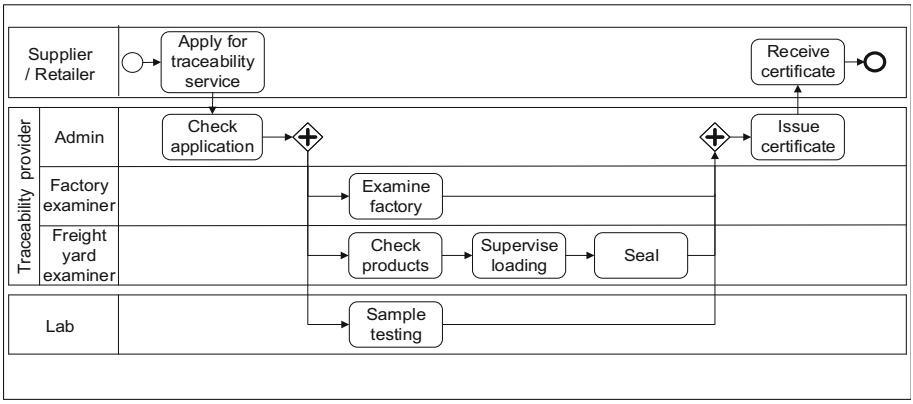


**Fig. 2.** Process of traceability services.

are met, the traceability company issues inspection certificates which represent the verification of quality and originality of products.

A real-world blockchain-based product traceability system called originChain was proposed in [4], in which the conventional centralised database is combined with decentralised blockchain technology. The supply chain industry requires data transparency and immutability to ensure the reliability of product information, which makes blockchain technology suitable for traceability system.

In originChain, we decided to store sensitive and small data on-chain, such as the hashes of certificates, the on-site freight yard photos and other traceability information like the result of sample testing, for data integrity, while the raw data are stored off-chain in the database. Applying some design patterns of smart contract to originChain can improve the quality of the whole system.

### 4.2 Contract Composer for Flexibility on Contract Instantiation

A *Contract Composer* instance is demonstrated in Listing 1.1. When a product supplier signs a legal agreement with traceability company, an on-chain version of the agreement should be created. An employee of the traceability company should input the basic information, including the names of the trading parties and the hash of agreement, and some required transaction information, such as price and valid period of each service selected according to the agreement. After inspection, the employee should call *confirm()* function, which can prevent revision to the data stored in this on-chain agreement. In addition, the address of the on-chain agreement should be appended into the actual legal agreement for data integrity.

For a contract that needs to contain many other objects, *Contract Composer* helps to improve flexibility when creating a contract instance. In a traceability process, factory examination, freight yard examination and sample testing are the optional services, and there is a *Service* interface in which a function is

defined but not implemented. For each service, there is a contract that implements the *Service* interface, overrides the abstract function, and stores the transaction information. In *LegalAgreement* contract, a service is initialised when the corresponding function is called, and after confirmation, the variable "confirmed" ensures that the data stored in the contract cannot be revised.

### 4.3 Contract Facade for Connectivity Between Contracts

Every batch of products needs contracts to store the hash of those sensitive traceability service data such as freight yard photos, sample test results, and inspection certificates, which should relate to the legal agreement of the corresponding companies, either. In Listing 1.2, a *FreightYardServiceFacade* contract is generated to connect the on-chain legal agreements generated by *Contract Composer* and the data contracts which store the traceability information. An originChain employee needs to provide batch number and address of on-chain legal agreement when calling function *link()*, in which a new instance of *FreightYardData* contract is created, the on-chain legal agreement address and the *FreightYardData* contract address are connected via the batch number. Similarly, other traceability services have their data contract too. There are accesses for the corresponding staffs of each traceability service process to upload the sensitive data. A freight yard examiner invokes *setFreightYardPic()* function to upload the hash of freight yard photos, while the smart contract records the address of that examiner automatically.

As mentioned in Sect. 3, here *Contract Facade* pattern helps connect two separate smart contracts. There can be more contracts linked by this pattern, nevertheless, connection of multiple contracts may result in a complicated facade contract, thus, developers should reach a balance in a facade contract according to their proximity.

### 4.4 Contract Observer for Updatability and Interoperability

A legal agreement needs to be revised when there are new regulations released by government. For example, the new Food Safety Law of China regulated new requirements on the formulation of national food safety standards and food safety traceability systems. Furthermore, it can be revised when the corresponding companies reach a new agreement on the traceability services. In either of the two cases, the on-chain legal agreement should be replaced by the new version. A direct way is to issue a new legal agreement contract and replace the old version address with the new one in all related contracts, however, it is tedious and unsafe to do the revision manually. *Contract Observer* pattern can deal with such problem.

Listing 1.3 demonstrates such a situation that the address of an old legal agreement needs to be updated to the new version. *Facade* is an interface, in which there is an abstract function *update()*. *FreightYardServiceFacade*, *FactoryServiceFacade* and *LabServiceFacade* are the contracts that connect legal agreement and the data contracts, they all implement the interface *Facade*.

```
interface Service{
  function setupContract(string sDate, string eDate, uint
      temPrice);
}
contract FactoryExamination is Service{
  string startingDate;
  string endingDate;
  uint price;
  function setupContract(string sDate, string eDate, uint
      temPrice){
    startingDate = sDate;
    endingDate = eDate;
    price = temPrice;}
  function getInfo constant returns (string, string, uint){
    return startingDate, endingDate, price;}
}
contract FreightYardExamination is Service{...}
contract SampleTesting is Service{...}
contract LegalAgreement{
  address firstParty;
  address secondParty;
  bytes32 contractHash;
  FactoryExamination FactoryService;
  FreightYardExamination FreightService;
  SampleTesting LabService;
  bool confirmed;
  function LegalContract(address firstP, address secondP,
      bytes32 cHash){
    if(!confirmed){
      firstParty = firstP;
      secondParty = secondP;
      contractHash = cHash;}
  }
  function setFactoryService(string temStart, string temEnd,
       uint temPrice){
    if(!confirmed){
      FactoryService = new FactoryExamination();
      FactoryService.setupContract(temStart, temEnd,
          temPrice);}
  }
  function setFreightService(string temStart, string temEnd,
       uint temPrice){...}
  function setLabService(string temStart, string temEnd,
      uint temPrice){...}
  function confirm(){
    confirmed = true;
  }
  ...
}
```

**Listing 1.1.** Contract composer.

*LegalAgreementObserver* is similar to the reception in a company, receiving information and notifying the corresponding departments. An originChain employee invokes *subscribe()* function to add the three *Facade Contracts* into "subscriber" in advance, and when the new on-chain legal agreement is created, the function *notify()* is called, to inform all the "subscriber" contract to update the legal agreement address.

```
contract LegalAgreement{...}
contract FreightYardData{
  bytes32 [] freightYardPic;
  address [] freightYardExaminer;
  function setFreightYardPic(bytes32 pic, address uploader){
      freightYardPic.push(pic);
      freightYardExaminer.push(uploader);
  }
  function getFreightYardPic(uint i) constant returns (
      bytes32, address){
      return (freightYardPic[i], freightYardExaminer[i]);
  }
  ...
}
interface Facade{...}
contract FreightYardServiceFacade is Facade{
  mapping (string => address) legalAgreement;
  mapping (string => address) traceData;
  mapping (address => string []) batchNo;
  function link(string batchID, address LegalAgreementAddr){
      legalAgreement[batchID] = LegalAgreementAddr;
      traceData[batchID] = new FreightYardData();
      batchNo[LegalContractAddr].push(batchID);
  }
  function setFreightYardPic(string batchID, bytes32 pic){
      FreightYardData(traceData[batchID]).setFreightYardPic(
          pic, msg.sender);
  }
  function getFreightYardPic(string batchID, uint i)
      constant returns (bytes32, address){
      return FreightYardData(traceData[batchID]).
          getFreightYardPic(i);
  }
  ...
}
```

**Listing 1.2.** Contract facade.

### 4.5 Hash Secret for Adaptability

A *Hash Secret* contract is shown in Listing 1.4. Every hash secret needs a struct to store hash key, while the boolean variable "init" can prevent the situation that a caller calls function *initial()* twice and then the previous hash key will

be revised without permission. The ciphertext of each hash secret should be generated off-chain but verified on-chain, which leads to the difference between the two parameters in function *changeKey()* and *verify()*.

*Hash Secret* can be used as both on-chain and off-chain component for permission control in a blockchain-based application system. For instance, when a batch of products needs sample testing service, *Hash Secret* can help with the dynamic binding of labs. An originChain employee initiates a hash secret and links it with his/her own address by invoking *initial()* function, then releases the address and plain-text of hash key to the available labs off-chain. Only the authorised labs can acquire "true" from the *verify()* function and upload the result of sample testing. As an on-line component, *Hash Secret* contract should be inherited or contained by another contract, and if it is used as an off-chain component, *Hash Secret* can interact with database or other components by *web3* library[5].

```
contract LegalAgreement{...}
...
interface Facade{
   function update(address oldAddr, address newAddr){};
}
contract FreightYardServiceFacade is Facade{
  mapping (string => address) legalAgreement;
  mapping (address => string []) batchNo;
  function update(address oldAddr, address newAddr){
    for(uint i = 0; i < batchNo[oldAddr].length; i++){
      legalAgreement[batchNo[oldAddr][i]] = newAddr;
      batchNo[newAddr].push(batchNo[oldAddr][i]);
    }
    delete batchNo[oldAddr];
  }
  ...
}
contract FactoryServiceFacade is Facade{...}
contract LabServiceFacade is Facade{...}
contract LegalAgreementObserver{
  address [] subscriber;
  function subscribe(address facadeAddr){
    subscriber.push(facadeAddr);
  }
  function notify(address oldAddr, address newAddr){
    for(uint i = 0; i < subscriber.length; i++)
      Facade(subscriber[i]).update(oldAddr, newAddr);
  }
}
```

**Listing 1.3.** Contract observer.

[5] https://github.com/ethereum/web3.js.

### 4.6 Multi-signature for Adaptability

*Multi-Signature*, similar to *Hash Secret* pattern, can act as both on-chain and off-chain component, Listing 1.5 mimics the multi-signature mechanism in Ethereum. In originChain, a client sends a request of issuing quality certificate, which requires the approval from traceability company, labs, and other related departments, then the result should be decided by these authorities.

The authorisation of *Hash Secret* is dynamic, while the authorities in *Multi-Signature* should be defined in advance. A request can be "agree request", aiming to the external business as issuing certificates, or "update request", aiming to the internal business of the authorities, such as changing authority address, and threshold of accepting a request. Different requests use different thresholds, but share the same pre-defined authority addresses. Each authority invokes the corresponding *Signature()* function to accept the request, and the request result is true if there are enough valid signatures. If so, further operations can be implemented. A requester can call *cancelAgreeRequest()* or *cancelUpdateRequest()* function to withdraw invalid request.

```
contract HashSecret{
  struct hashSecret{
    bytes32 hashKey;
    bool init;
  }
  mapping (address => hashSecret) secret;
  function initial(bytes32 key){
    if(secret[msg.sender].init != true){
      secret[msg.sender].hashKey = key;
      secret[msg.sender].init = true;}
  }
  function changeKey(string oldKey, bytes32 newKey){
    if(secret[msg.sender].init == true)
      if(secret[msg.sender].hashKey == sha256(oldKey))
        secret[msg.sender].hashKey = newKey;
}
  function verify(address initiator, string inputKey)
      constant returns (bool){
    if(secret[initiator].hashKey == sha256(inputKey))
      return true;
    else
      return false;
    }
}
```

**Listing 1.4.** Hash secret.

## 5 Discussion

The patterns are divided into four categories in this study to improve the non-functional requirements of blockchain-based applications, and in this section we

share some experiences that learn from applying several proposed patterns to a blockchain-based application.

```
contract multiSignature{
  uint total;
  address[] authorities;
  uint agreeThreshold;
  uint updateThreshold;
  address agreeRequester;
  address updateRequester;
  mapping(address => bool) agreeState;
  mapping(address => bool) updateState;
  function multiSignature(uint totalAut, uint aThres, uint
      uThres, address[] aut){...}
  function agreeSignature(){
    agreeState[msg.sender] = true;
  }
  function agreeResult() returns (bool){
    uint k = 0;
    for(uint i = 0; i < total; i++){
      if(agreeState[authorities[i]] == true)
          k++;
    }
    if(k >= agreeThreshold)
      return true;
    else
      return false;
  }
  function initialAgree() internal{...}
  function cancelAgreeRequest(){
    if(msg.sender == agreeRequester)
      initialAgree();
  }
  function updateSignature(){...}
  function updateResult() returns (bool){...}
  function initialUpdate() internal{...}
  function updateAuthorityList(uint newTotal, address[] par)
      {...}
  function updateAgreeThreshold(uint newThres){...}
  function cancelUpdateRequest(){...}
  ...
}
```

**Listing 1.5.** Multi-signature.

**Interoperation Among Smart Contracts.** *Creational Pattern*, *Structural Pattern* and *Inter-Behavioral Pattern* focus on the structural design of smart contracts. Contracts are interactive when applying these patterns, for example, in Sect. 4.4, interface *Facade* is implemented by three facade contracts, all of which contain the addresses and invocations of *LegalAgreement*.

*LegalAgreementObserver* contains the addresses of the three facade contracts, and invokes them via interface *Facade*.

Smart contracts inherit, contain, or invoke each other, which enriches the architectural design, when applied these three kinds of patterns. *Creational Pattern* concentrates on the instantiation of smart contracts, *Structural Pattern* helps manage the relationship among contracts, *Inter-Behavioral Pattern* enhances the flexibility when contract instances operating with each other.

However, the issue of a structured contract includes more complex permission control and cost. Applying such patterns brings the consequence that the architecture of a smart contract may be complicated if it clutters an excess of functions. Moreover, it is hard to manage permission control as a complicated contract may involve multiple roles. Developers need to make reasonable separation when designing smart contracts to balance the coupling degree.

**Independence of Intra-behavioral Patterns.** Every *Intra-Behavioral Pattern* can work both individually or with other contracts as they do not need to rely on other smart contracts, namely, they do not need to inherit, contain or invoke other contracts but the contrary. Taking *Hash Secret* as an example, it provides a flexible way to implement permission control, either off-chain or on-chain. In Fig. 1, *Hash Secret* verifies a network participant's identity and sends the result to server, only the authorised ones can continue to further operations. Apart from that, smart contracts can inherit or embody this contract to obtain the ability of on-chain permission control.

**Reusable Logic Saves Storage.** Storage is one of the limitations of blockchain technology. According to blockchain's properties, every participant has a local replica of the whole transaction history, moreover, any revision or deletion to the existing transactions is not allowed, thus joining a blockchain network can be a heavy burden to individual's storage. Design patterns provide smart contracts reusable logic, which helps relieve such burden. For example, developers can change the secret key in *Hash Secret*, and revoke the request or update the authority information in *Multi-Signature* on demand to achieve reusability. Besides, *Contract Decorator* aims to encapsulate the old contracts into the new ones, which can also be considered as a method of saving storage.

## 6 Conclusion and Future Work

In this paper, we propose a taxonomy of the design patterns for smart contracts, and share the experiences of applying several patterns of *Solidity*-based smart contract to a real-world blockchain-based traceability system called originChain. Blockchain's unique properties provide new thoughts to application architecture, in which blockchain technology acts as a special component. The design patterns affect some specific aspects of the blockchain-based application, such as updatability, adaptability and interoperability.

We divide the design patterns into four categories: *Creational Pattern*, *Structural Pattern*, *Inter-Behavioral Pattern* and *Intra-Behavioral Pattern*, according to their contribution to the architecture design of smart contracts. Specifically, we apply *Contract Composer*, *Contract Facade*, *Contract Observer*, *Hash Secret*, and *Multi-Signature* to a real-world blockchain-based application to improve the non-functional requirements.

Smart contracts used to be standalone and aimed at a specific function, but now in a blockchain-based application, smart contracts can cope with some complex business process instead of barely store the data. Applying some software design patterns to smart contracts helps developers to have a better design on the architecture of smart contracts.

The future work includes summarising more design patterns of smart contract, refining the taxonomy, and giving more detailed discussion about the patterns. In addition, we will optimise the blockchain-based traceability system and extend blockchain technology to other potential industries.

## References

1. Bartoletti, M., Pompianu, L.: An empirical analysis of smart contracts: platforms, applications, and design patterns. ArXiv e-prints, March 2017
2. Eberhardt, J., Tai, S.: On or off the blockchain? Insights on off-chaining computation and data. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) ESOCC 2017. LNCS, vol. 10465, pp. 3–15. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67262-5_1
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Elements of Reusable Object-oriented Software. Pearson Education, London (1995)
4. Lu, Q., Xu, X.: Adaptable blockchain-based systems: a case study for product traceability. IEEE Softw. **34**(6), 21–27 (2017)
5. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
6. Omohundro, S.: Cryptocurrencies, smart contracts, and artificial intelligence. AI Matters **1**(2), 19–21 (2014). https://doi.org/10.1145/2685328.2685334
7. Staples, M., Chen, S., Falamaki, S., Ponomarev, A., Rimba, P., Weber, A.B.T.I., Xu, X., Zhu, J.: Risks and opportunities for systems using blockchain and smart contracts. Technical report, Sydney (2017). Data61(CSIRO)
8. Swan, M.: Blockchain: Blueprint for a New Economy. O'Reilly, Sebastopol (2015)
9. Weber, I., Xu, X., Riveret, R., Governatori, G., Ponomarev, A., Mendling, J.: Untrusted business process monitoring and execution using blockchain. In: La Rosa, M., Loos, P., Pastor, O. (eds.) BPM 2016. LNCS, vol. 9850, pp. 329–347. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45348-4_19
10. Xu, X., Pautasso, C., Zhu, L., Gramoli, V., Ponomarev, A., Tran, A.B., Chen, S.: The blockchain as a software connector. In: The 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), Venice, Italy (2016)
11. Xu, X., Webber, I., Staples, M., et al.: A taxonomy of blockchain-based systems for architecture design. In: IEEE International Conference on Software Architecture (ICSA), Gothenburg, Sweden (2017)
12. Zhang, P., White, J., Schmidt, D.C., Lenz, G.: Applying Software Patterns to Address Interoperability in Blockchain-based Healthcare Apps. ArXiv e-prints, June 2017