2018 IEEE Confs on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, Congress on Cybermatics

Design Patterns for Smart Contracts in the Ethereum Ecosystem

Maximilian Wöhrer and Uwe Zdun
University of Vienna
Faculty of Computer Science
Währingerstraße 29, 1090 Vienna, Austria
Email: {maximilian.woehrer,uwe.zdun}@univie.ac.at

***Abstract*—The idea to digitally facilitate contract law and business practices through computer programs has led to the notion of smart contracts. Today's most prominent smart contract ecosystem is Ethereum, a blockchain based distributed computing platform. 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. Based on a Multivocal Literature Research and an analysis of the gathered data based on qualitative research methods, we mined a number of design patterns providing design guidelines. We describe those patterns in detail and provide exemplary code for better illustration. Our research shows that the patterns are widely used to address application requirements and common problems. We expect generalizability of some or all of the patterns for other smart contract ecosystems, but this is outside of the scope of this study, which studied only smart contract patterns in Ethereum.**

I. Introduction

Bitcoin, which is the most popular cryptocurrency, records transactions in a decentralized data structure called blockchain and supports the feature to encode rules or simple scripts for processing transactions. This feature has evolved to the concept of smart contracts, self-executing computer programs that run on a blockchain to stipulate and enforce the negotiation and execution of (legal) contracts. The blockchain, or more precisely its decentralized nature, assures that contract initiated transactions are autonomously and truthfully executed. Today's most prominent smart contract ecosystem is Ethereum, a blockchain based distributed computing platform, allowing anyone to write smart contracts with arbitrary rules in the platform's leading language Solidity. Despite the increasing popularity of smart contracts, their implementation involves a number of problems. First, rather unconventional programming paradigms are required, because of the inherent characteristics of blockchain-based program execution. For example, programmers have to consider the lack of execution control and the immutable character of smart contracts once they are deployed. Second, due to missing low-level programming abstractions the developer is responsible for the internal organization and manipulation of data at a deeper level. Third, the rapid transformation of platform features and security considerations requires continuous awareness of platform capabilities and potential security risks. Furthermore, smart contracts handle considerable financial values, therefore it is crucial that their implementation is correct and secure against attacks. Given these points, it is beneficial to have a solid foundation of established design and coding guidelines that promote the creation of correct and secure smart contracts, for example in the form of design patterns. Design patterns [1, 2] are a commonly used technique to encode design guidelines or best practices. They express an abstract or conceptual solution to a concrete, complex, and reoccurring problem. So far, design patterns have not received a lot of attention in Ethereum research and information on Solidity design and coding guidelines is scattered among different sources. In previous work [3] we have gathered security related design patterns. In this work, we focus on general design patterns for smart contracts in Ethereum. Our research aims to answer the following two research questions (RQs):

RQ1 Which design patterns commonly appear in the Ethereum ecosystem?

RQ2 How do these design patterns map to Solidity coding practices?

In order to answer these questions, we followed the Multivocal Literature Research method by Garousi et al. [4] to incorporate practitioners' experience and applied an analysis of the gathered data based on qualitative research methods (namely Grounded Theory [5] techniques to synthesize the patterns). Our research identified several patterns that pinpoint common issues during the implementation of smart contracts and provide guidance to resolve them.

The paper is organised in the following way: First, we discuss the research study design in Section II, before we present design patterns for Solidity in Section III, and discuss our findings in Section IV. Finally, we present related work in Section V, and draw a conclusion in Section VI.

II. Research Study Design

Due to a lack of academic literature regarding design patterns for Ethereum and Solidity we decided to carry out a Multivocal Literature Review (MLR). A MLR is a form of Systematic Literature Review (SLR) which includes "grey" literature (e.g., blogs, videos, and web pages) in addition to published "white" literature (e.g., academic journals, and conference papers) [6]. Figure 1 depicts the general process of our conducted MLR incorporating guidelines elaborated by Garousi et al. [6]. Starting from our research questions

978-1-5386-7975-3/18/$31.00 ©2018 IEEE
DOI 10.1109/Cybermatics_2018.2018.00255

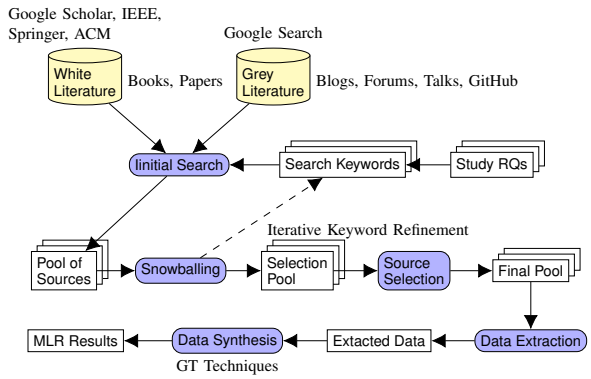

Fig. 1. An overview of the conducted Multivocal Literature Review (MLR) process.

we defined initial search keywords as "ethereum", "solidity", "(smart) contract", and "(software/design) pattern". These keyword combinations were then used to query different data sources for "white" and "grey" literature. The results were examined, i.e., citations and links were followed and reference lists were studied during a process called snowballing [7]. At the same time, initial search keywords were iteratively extended until theoretical saturation was reached. Next, the subsequent pool of sources was filtered according to predefined inclusion and exclusion criteria which encompassed to accept sources of any type that relate to Ethereum design patterns and exclude non-English works or works that seem unbalanced in presentation. Further, as Ethereum and Solidity have significantly evolved in recent years, we prioritized recent works. The accumulated final source pool contained among others the following important major sources. First, academic literature related to Ethereum and Solidity patterns [8, 9, 10]. Second, the official Solidity development documentation [11] and smart contract best practices [12]. Third, Internet blogs and discussion forums about Ethereum, such as the Ethereum community on reddit [13], and the Ethereum QA section on stackexchange [14]. Forth, Ethereum conference talks [15, 16]. Fifth, existing GitHub repositories related to smart contract coding patterns in Solidity [17, 18, 19]. As next step the final source pool was reviewed and the extracted relevant information was analysed with Grounded Theory (GT) techniques, following recommendations by Stol et al. [5]. In general, we took an iterative and pragmatic approach and recorded the concepts of our observations and insights in theoretical memos. These memos represented the actual pattern synthesis process and happened in several iterative stages, in which the patterns were constantly compared, revised, and contrasted until all the gathered information was accounted for.

III. Smart Contract Design Patterns

In this section we give an overview of design patterns that are notably frequent or practical for smart contract design and coding. The patterns are divided according to operational scope into five categories: Action and Control (III-A), Authorization (III-B), Lifecycle (III-C), Maintenance (III-D), and Security [3]. An overview of the categories and assigned patterns is given in Table I, which also lists an example contract with published source code for each pattern on the Ethereum mainnet. The illustrative source code examples for the patterns in this section are available on GitHub [20].

TABLE I
Pattern overview including example contracts with published source code on the Ethereum mainnet.

Category	Pattern	Example Contract
Action and Control	Pull Payment	Cryptopunks
	State Machine	DutchAuction
	Commit and Reveal	ENS Registrar
	Oracle (Data Provider)	Etheroll
Authorization	Ownership	Ethereum Lottery
	Accesss Restriction	Etheroll
Lifecycle	Mortal	GTA Token
	Automatic Deprecation	Polkadot
Maintenance	Data Segregation	SAN Token
	Satellite	LATP Token
	Contract Register	Tether Token
	Contract Relay	Numeraire
Security[3]	Checks-Effects-Interaction	CryptoKitties
	Emergency Stop	Augur/REP
	Speed Bump	TheDAO
	Rate Limit	etherep
	Mutex	Ventana Token
	Balance Limit	CATToken

A. Action and Control Patterns

Action and Control is a group of patterns that provide mechanisms for typical operational tasks.

1) Pull Payment:

PULL PAYMENT PATTERN
Problem When a contract sends funds to another party, the send operation can fail.
Solution Let the receiver of a payment withdraw the funds.

A common task when coding smart contracts is to transfer funds. Unfortunately, there are several circumstances under which a transfer can fail. This is due to the fact that the implementation to send funds involves an external call, which basically hands over control to the called contract. Therefore, security considerations regarding external calls and re-entrancy attacks have to be considered. A re-entrancy attack describes the scenario where the called contract calls back the current contract, before the first invocation of the function containing the call, was finished. This can lead to an unwanted execution behaviour of functions.

Currently, there are three different methods to transfer funds in Solidity. These are `address.send()`, `address.transfer()`, and `address.call.value()()`. If the payment recipient is a contract, calling theses methods triggers the execution of a so-called fallback function in the receiver

contract. Per definition, the fallback function is a name- and parameterless function, that is called when the function signature does not match any of the available functions in a Solidity contract. Since `send()` specifies a blank function signature, it will always trigger the fallback function if it exists. `x.transfer(y)` is equivalent to `require(x.send(y))` and defines a maximum stipend of 2,300 gas, given to the receiver contract for execution, which is currently only enough to log an event. `address.call.value()()` gives all available gas to the receiving contract for execution, which makes this type of value transfer unsafe against re-entrancy. So, the difference between `send()` and `address.call.value()()` is how much gas is made available to the fallback function in the receiving contract, thereby controlling the risk.

Due to the possibility of deliberately sabotaging the transfer of funds by executing expensive operations in the fallback method, causing an "out of gas" (OOG) error, or manipulations involving re-entrancy attacks, a more favourable approach is to reverse the payment process (let users withdraw their funds themselves). Listing 1 shows a problematic reliance on a successful transfer of funds, whereas Listing 2 mitigates this problem by isolating the external call into its own transaction that can be initiated by the recipient of the call.

```
pragma solidity ^0.4.17;
contract Auction {
  address public highestBidder;
  uint highestBid;

  function bid() public payable {
    require(msg.value >= highestBid);
    if (highestBidder != 0) {
      // if call fails causing a rollback,
      // no one else can bid
      highestBidder.transfer(highestBid);
    }
    highestBidder = msg.sender;
    highestBid = msg.value;
  }
}
```

Listing 1. An intuitive solution in an auction contract would be to push a payment to a defeated bidder once a higher bid has been received.

```
pragma solidity ^0.4.17;
contract Auction {
  address public highestBidder;
  uint highestBid;
  mapping(address => uint) refunds;

  function bid() public payable {
    require(msg.value >= highestBid);
    if (highestBidder != 0) {
      // record the underlying bid to be refund
      refunds[highestBidder] += highestBid;
    }
    highestBidder = msg.sender;
    highestBid = msg.value;
  }

  function withdrawRefund() public {
    uint refund = refunds[msg.sender];
    refunds[msg.sender] = 0;
    msg.sender.transfer(refund);
  }
}
```

Listing 2. Introducing a refunds mapping, which stores the claimable defeated bids, to be withdrawn by participants in a pull payment process.

2) State Machine:

STATE MACHINE PATTERN
Problem An application scenario implicates different behavioural stages and transitions.
Solution Apply a state machine to model and represent different behavioural contract stages and their transitions.

A state machine models the behaviour of a system based on its history and current inputs. Developers use this construct to break complex problems into simple states and state transitions. These are then used to represent and control the execution flow of a program. State machines can also be applied in smart contracts, exemplified in Listing 3. Many usage scenarios require a contract to have different behavioural stages, in which different functions can be called. When interacting with such a contract, a function call might end the current stage and initiate a change into a consecutive stage.

```
pragma solidity ^0.4.17;
contract DepositLock {
  enum Stages {
    AcceptingDeposits,
    FreezingDeposits,
    ReleasingDeposits
  }
  Stages public stage = Stages.AcceptingDeposits;
  uint public creationTime = now;
  mapping (address => uint) balances;

  modifier atStage(Stages _stage) {
    require(stage == _stage);
    _;
  }

  modifier timedTransitions() {
    if (stage == Stages.AcceptingDeposits && now >=
        creationTime + 1 days)
      nextStage();
    if (stage == Stages.FreezingDeposits && now >=
        creationTime + 8 days)
      nextStage();
    _;
  }

  function nextStage() internal {
    stage = Stages(uint(stage) + 1);
  }

  function deposit() public payable timedTransitions
      atStage(Stages.AcceptingDeposits) {
    balances[msg.sender] += msg.value;
  }

  function withdraw() public timedTransitions atStage(
      Stages.ReleasingDeposits) {
    uint amount = balances[msg.sender];
    balances[msg.sender] = 0;
    msg.sender.transfer(amount);
  }
}
```

Listing 3. A contract based on a state machine to represent a deposit lock, which accepts deposits for a period of one day and releases them after seven days.

3) Commit and Reveal:

COMMIT AND REVEAL PATTERN
Problem All data and every transaction is publicly visible on the blockchain, but an application scenario requires that contract interactions, specifically submitted parameter values, are treated confidentially.
Solution Apply a commitment scheme to ensure that a value submission is binding and concealed until a consolidation phase runs out, after which the value is revealed, and it is publicly verifiable that the value remained unchanged.

A characteristic of blockchains is, that it is not possible to restrict anyone from reading contents of a transaction or transaction's state. This transparency leads to problems, especially when contract participants compete with each other.

```
pragma solidity ^0.4.17;
contract CommitReveal {
  struct Commit {string choice; string secret; string
      status;}
  mapping(address => mapping(bytes32 => Commit)) public
      userCommits;

  event LogCommit(bytes32, address);
  event LogReveal(bytes32, address, string, string);

  function CommitReveal() public {}

  function commit(bytes32 _commit) public returns (bool
      success) {
    var userCommit = userCommits[msg.sender][_commit];
    if(bytes(userCommit.status).length != 0) {
      return false; // commit has been used before
    }
    userCommit.status = "c"; // comitted
    LogCommit(_commit, msg.sender);
    return true;
  }

  function reveal(string _choice, string _secret, bytes32
      _commit) public returns (bool success) {
    var userCommit = userCommits[msg.sender][_commit];
    bytes memory bytesStatus = bytes(userCommit.status);
    if(bytesStatus.length == 0) {
      return false; // choice not committed before
    } else if (bytesStatus[0] == "r") {
      return false; // choice already revealed
    }
    if (_commit != keccak256(_choice, _secret)) {
      return false; // hash does not match commit
    }
    userCommit.choice = _choice;
    userCommit.secret = _secret;
    userCommit.status = "r"; // revealed
    LogReveal(_commit, msg.sender, _choice, _secret);
    return true;
  }

  function traceCommit(address _address, bytes32 _commit)
      public view returns (string choice, string secret,
      string status) {
    var userCommit = userCommits[_address][_commit];
    require(bytes(userCommit.status)[0] == "r");
    return (userCommit.choice, userCommit.secret,
        userCommit.status);
  }
}
```

Listing 4. A contract that allows a party to commit to a choice and reveal it at a later point in time, traceable for anyone.

4) *Oracle (Data Provider):*

ORACLE (DATA PROVIDER) PATTERN
Problem An application scenario requires knowledge contained outside the blockchain, but Ethereum contracts cannot directly acquire information from the outside world. On the contrary, they rely on the outside world pushing information into the network.
Solution Request external data through an oracle service that is connected to the outside world and acts as a data carrier.

Ethereum contracts run within their own ecosystem, where they communicate with each other, but external data can only enter the system through outside interaction via a transaction (by passing data to a method). This is a drawback, because many contract use cases depend on external knowledge outside the blockchain (e.g. price feeds). A solution to this problem is to utilize oracles with a connection to the outside world. The oracle service acts as a data carrier, where the interaction between an oracle service and a smart contract is asynchronous. First, a transaction invokes a function of a smart contract that contains an instruction to send a request to an oracle. Then, according to the parameters of such a request, the oracle will fetch a result and return it by executing a callback function placed in the primary contract. The described procedure involving an oracle contract and its consumer contract is illustrated by Listing 5 and Listing 6. A shortcoming of this solution is that oracles contradict the blockchain theorem of a decentralized network, because contracts utilizing an oracle rely on a single party or group to be honest. Currently operating oracle services [21, 22] address this shortcoming by accompanying the resulting data with a proof of authenticity. It should be noted, that an oracle has to pay for the callback invocation, thus an oracle usually requires payment of an oracle fee, plus Ether necessary for the callback.

```
pragma solidity ^0.4.17;
contract Oracle {
  address knownSource = 0x123...; // known source
  struct Request {
    bytes data;
    function(bytes memory) external callback;
  }
  Request[] requests;

  event NewRequest(uint);

  modifier onlyBy(address account) {
    require(msg.sender == account); _;
  }

  function query(bytes data, function(bytes memory)
      external callback) public {
    requests.push(Request(data, callback));
    NewRequest(requests.length - 1);
  }

  // invoked by outside world
  function reply(uint requestID, bytes response) public
      onlyBy(knownSource) {
    requests[requestID].callback(response);
  }
}
```

Listing 5. An oracle contract that allows to request data from outside the blockchain.

```
pragma solidity ^0.4.17;
import "./Oracle.sol";
contract OracleConsumer {
  Oracle oracle = Oracle(0x123...); // known contract

  modifier onlyBy(address account) {
    require(msg.sender == account); _;
  }

  function updateExchangeRate() {
    oracle.query("USD", this.oracleResponse);
  }

  function oracleResponse(bytes response) onlyBy(oracle) {
    // use the data
  }
}
```

Listing 6. An oracle consumer contract implementing a callback method to receive result data.

B. *Authorization Patterns*

Authorization is a group of patterns that control access to smart contract functions and provide basic authorization control, which simplify the implementation of "user permissions".

1) Ownership:

OWNERSHIP PATTERN
Problem By default any party can call a contract method, but it must be ensured that sensitive contract methods can only be executed by the owner of a contract.
Solution Store the contract creator's address as owner of a contract and restrict method execution dependent on the callers address.

It is very common that only the owner of a contract should be eligible to call functions, which are sensitive and crucial for the correct operation of the contract. This pattern limits access to certain functions to only the owner of the contract, an example is shown in Listing 7. A typical application of this pattern is demonstrated in the Mortal pattern.

```
pragma solidity ^0.4.17;
contract Owned {
  address public owner;

  event LogOwnershipTransferred(address indexed
      previousOwner, address indexed newOwner);

  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }

  function Owned() public {
    owner = msg.sender;
  }

  function transferOwnership(address newOwner) public
      onlyOwner {
    require(newOwner != address(0));
    LogOwnershipTransferred(owner, newOwner);
    owner = newOwner;
  }
}
```

Listing 7. A simple contract to track the ownership of a contract.

2) Access Restriction:

ACCESS RESTRICTION PATTERN
Problem By default a contract method is executed without any preconditions being checked, but it is desired that the execution is only allowed if certain requirements are met.
Solution Define generally applicable modifiers that check the desired requirements and apply these modifiers in the function definition.

Since there is no built in mechanism to control execution privileges, a common pattern is to restrict function execution. It is often required that functions should only be executed based on the presence of certain prerequisites. These can refer to different categories, such as temporal conditions, caller and transaction info, or other requirements that need to be checked prior a function execution. Listing 8 illustrates how different prerequisites can be checked prior function execution.

```
pragma solidity ^0.4.17;
import "./Ownership.sol";
contract AccessRestriction is Owned {
  uint public creationTime = now;

  modifier onlyBefore(uint _time) {
    require(now < _time); _;
  }

  modifier onlyAfter(uint _time) {
    require(now > _time); _;
  }

  modifier onlyBy(address account) {
    require(msg.sender == account); _;
  }

  modifier condition(bool _condition) {
    require(_condition); _;
  }

  modifier minAmount(uint _amount) {
    require(msg.value >= _amount); _;
  }

  function f() payable onlyAfter(creationTime + 1 minutes)
      onlyBy(owner) minAmount(2 ether) condition(msg.
      sender.balance >= 50 ether) {
    // some code
  }
}
```

Listing 8. A contract demonstrating how to check certain requirements prior to function execution.

C. Lifecycle Patterns

Lifecycle is a group of patterns that control the creation and destruction of smart contracts.

1) Mortal:

MORTAL PATTERN
Problem A deployed contract will exist as long as the Ethereum network exists. If a contract's lifetime is over, it must be possible to destroy a contract and stop it from operating.
Solution Use a selfdestruct call within a method that does a preliminary authorization check of the invoking party.

A contract is defined by its creator, but the execution, and subsequently the services it offers are provided by the Ethereum network itself. Thus, a contract will exist and be executable as long as the whole network exists, and will only disappear if it was programmed to self destruct. Mortal is a pattern that enables the creator of a contract to destroy it. The pattern uses a modifier to ensure that only the owner of the contract can execute the `selfdestruct` operation, which sends the remaining Ether stored within the contract to a designated target address (provided as argument) and then the storage and code is cleared from the current state. Listing 9 exemplifies the application of this pattern.

```
pragma solidity ^0.4.17;
import "../authorization/Ownership.sol";
contract Mortal is Owned {
  function destroy() public onlyOwner {
    selfdestruct(owner);
  }

  function destroyAndSend(address recipient) public
      onlyOwner {
    selfdestruct(recipient);
  }
}
```

Listing 9. A contract that provides its creator with the ability to destroy it.

2) Automatic Deprecation:

AUTOMATIC DEPRECATION PATTERN
Problem A usage scenario requires a temporal constraint defining a point in time when functions become deprecated.
Solution Define an expiration time and apply modifiers in function definitions to disable function execution if the expiration date has been reached.

Automatic deprecation is a pattern that allows to automatically prohibit the execution of functions after a specific time

period has elapsed. Listing 10 shows the automatic deprecation of functions based on an elapsed time period.

```
pragma solidity ^0.4.17;
contract AutoDeprecate {
  uint expires;

  function AutoDeprecate(uint _days) public {
    expires = now + _days * 1 days;
  }

  function expired() internal view returns (bool) {
    return now > expires;
  }

  modifier willDeprecate() {
    require(!expired());
    _;
  }

  modifier whenDeprecated() {
    require(expired());
    _;
  }

  function deposit() public payable willDeprecate {
    // some code
  }

  function withdraw() public view whenDeprecated {
    // some code
  }
}
```

Listing 10. A contract interface that automatically deprecates after a specified time period has elapsed.

D. Maintenance Patterns

Maintenance is a group of patterns that provide mechanisms for live operating contracts. In contrast to ordinary distributed applications, which can be updated when bugs are detected, smart contracts are irreversible and immutable. This means that there is no way to update a smart contract, other than writing an improved version that is then deployed as new contract.

1) Data Segregation:

DATA SEGREGATION PATTERN
Problem Contract data and its logic are usually kept in the same contract, leading to a closely entangled coupling. Once a contract is replaced by a newer version, the former contract data must be migrated to the new contract version.
Solution Decouple the data from the operational logic into separate contracts.

The data segregation pattern separates contract logic from its underlying data. Segregation promotes the separation of concerns and mimics a layered design (e.g. logic layer, data layer). Following this principle avoids costly data migrations when code functionality changes. Meaning a new contract version would not have to recreate all of the existing data contained in the previous contract. The separation of contract data and contract logic is shown in Listing 11 and Listing 12. It is favourable to design the storage contract very generic so that once it is created, it can store and access different types of data with the help of setter and getter methods.

```
pragma solidity ^0.4.17;
contract DataStorage {
  mapping(bytes32 => uint) uintStorage;

  function getUintValue(bytes32 key) public constant
      returns (uint) {
    return uintStorage[key];
  }

  function setUintValue(bytes32 key, uint value) public {
    uintStorage[key] = value;
  }
}
```

Listing 11. The data is separated in its own contract.

```
pragma solidity ^0.4.17;
import "./DataStorage.sol";
contract Logic {
  DataStorage dataStorage;

  function Logic(address _address) public {
    dataStorage = DataStorage(_address);
  }

  function f() public {
    bytes32 key = keccak256("emergency");
    dataStorage.setUintValue(key, 911);
    dataStorage.getUintValue(key);
  }
}
```

Listing 12. The contract logic can manipulate the data through a reference.

2) Satellite:

SATELLITE PATTERN
Problem Contracts are immutable. Changing contract functionality requires the deployment of a new contract.
Solution Outsource functional units that are likely to change into separate so-called satellite contracts and use a reference to these contracts in order to utilize needed functionality.

The satellite pattern allows to modify and replace contract functionality. This is achieved through the creation of separate satellite contracts that encapsulate certain contract functionality. The addresses of these satellite contracts are stored in a base contract. This contract can then can call out to the satellite contracts when it needs to reference certain functionalities, by using the stored address pointers. If this pattern is properly implemented, modifying functionality is as simple as creating new satellite contracts and changing the corresponding satellite addresses. Listing 13 and 14 exemplify the application of this pattern.

```
pragma solidity ^0.4.17;
contract Satellite {
  function calculateVariable() public pure returns (uint){
    // calculate var
    return 2 * 3;
  }
}
```

Listing 13. A satellite contract encapsulates certain contract functionalities.

```
pragma solidity ^0.4.17;
import "../../authorization/Ownership.sol";
import "./Satellite.sol";
contract Base is Owned {
  uint public variable;
  address satelliteAddress;

  function setVariable() public onlyOwner {
    Satellite s = Satellite(satelliteAddress);
    variable = s.calculateVariable();
  }

  function updateSatelliteAddress(address _address) public
      onlyOwner {
```

```
    satelliteAddress = _address;
  }
}
```

Listing 14. A base contract referring to a satellite contract in order to fulfil its purpose. The use of a satellite allows an easy contract functionality modification.

3) Contract Register:

CONTRACT REGISTER PATTERN
Problem Contract participants must be referred to the latest contract version.
Solution Let contract participants pro-actively query the latest contract address through a register contract that returns the address of the most recent version.

The register pattern is an approach to handle the update process of a contract. The pattern keeps track of different versions (addresses) of a contract and points on request to the latest one, as seen in Listing 15. In conclusion, before interacting with a contract, a user would always have to query the register for the contract's latest address. When following this update approach, it is also important to determine how to handle existing contract data, when an old contract version is replaced. An alternative solution to point to the latest contract address would be to utilize the Ethereum Name Service (ENS). It is a register that enables a secure and decentralised way to resolve human-readable names, like 'mycontract.eth', into machine-readable identifiers, including Ethereum addresses.

```
pragma solidity ^0.4.17;
import "../authorization/Ownership.sol";
contract Register is Owned {
  address backendContract;
  address[] previousBackends;

  function Register() public {
    owner = msg.sender;
  }

  function changeBackend(address newBackend) public
       onlyOwner() returns (bool) {
    if(newBackend != backendContract) {
      previousBackends.push(backendContract);
      backendContract = newBackend;
      return true;
    }
    return false;
  }
}
```

Listing 15. A register contract to store the latest version of a contract.

4) Contract Relay:

CONTRACT RELAY PATTERN
Problem Contract participants must be referred to the latest contract version.
Solution Contract participants always interact with the same proxy contract that relays all requests to the most recent contract version.

A relay is another approach to handle the update process of a contract. The relay pattern provides a method to update a contract to a newer version while keeping the old contract address. This is achieved by using a kind of proxy contract that forwards calls and data to the latest version of the contract, shown in Listing 16. This approach can forward function calls including their arguments, but cannot return result values. Another drawback of this approach is that the data storage layout needs to be consistent in newer contract versions, otherwise data may be corrupted.

```
pragma solidity ^0.4.17;
import "../authorization/Ownership.sol";
contract Relay is Owned {
  address public currentVersion;

  function Relay(address initAddr) public {
    currentVersion = initAddr;
    owner = msg.sender;
  }

  function changeContract(address newVersion) public
       onlyOwner() {
    currentVersion = newVersion;
  }

  // fallback function
  function() public {
    require(currentVersion.delegatecall(msg.data));
  }
}
```

Listing 16. A relay contract to forward data and calls.

IV. Discussion

Combined with our previous work [3] our research covers 18 patterns grouped into five categories. An examination of the patterns reveals a hierarchy structure, meaning some of the patterns act as foundation for others. For example, the Access Restriction pattern, is directly applied by other patterns, like the Mortal pattern. Further, although some patterns are very basic, their real practical value is unfolded when patterns are combined. For example, the Ownership pattern is often used as a prerequisite in combination with other patterns. A principle shared by several patterns is related to the problem of contract immutability, which is circumvented by using updatable object references. All Maintenance Patterns use this principle to decouple contract functionality, data, or even whole contracts through a proxy object. As to the generalizability of the patterns, it might be assumed that other platforms face similar issues as Ethereum. In the real world, once a contract is changed, it needs to be revalidated by all involved parties. This concept is also encountered in Ethereum, where contracts are immutable and any change requires the creation of a new contract. Although real world contracts are conclusive and final through their written terms, their code implementations underlie inherent software concepts involving evolutionary code changes and bug fixes. This creates a divergence between contract immutability (a final version of a written agreement manifested in code) and the ability to modify that code (due to bugs or a necessary code updates). That is, the separation of code changes that modify contract terms and those that are necessary due to evolutionary adaptations is important. Altogether, because any software based smart contract ecosystem and its contained contracts require code updates, it can be assumed that maintenance patterns are generally applicable to other ecosystems as well.

V. Related Work

According to Alharby and van Moorsel [23] current research on smart contracts is focused on identifying and tackling smart

contract issues and can be divided into four categories, namely coding, security, privacy and performance issues. Unfortunately, a lot of research and practical knowledge is scattered throughout grey literature. Only few papers mention software patterns in blockchain technology. A work with general scope on blockchain software development written by Xu et al. [24] proposes a taxonomy of blockchain-based systems on architecture design. Another work by Bartoletti and Pompianu [8] conducted an empirical analysis of Solidity contracts and identified a list of nine common design patterns. Yet another paper by Zhang et al. [9] describes how the application of familiar software patterns can help to resolve design specific challenges. Finally, a paper by Mavridou and Laszka [10] describes a framework for designing contracts as Finite State Machines utilising design patterns for code generation. In general, the above mentioned papers have commonalities with this work. Some findings and principles intersect with the patterns in this paper, but none of these publications focus on a dedicated pattern language to the extent of this work.

VI. Conclusion

In this paper we derived Solidity design patterns from white and grey literature using a Multivocal Literature Review and qualitative research methods borrowed from Grounded Theory. While many smart contracts have been written in Solidity for different purposes, we have identified, grouped, and described several globally applicable patterns and have discussed common principles and relationships among them. Each pattern is explained in a problem and solution based approach, to illustrate the context and applicability of the pattern. With our work, we aim to provide a basis for a comprehensive pattern language, that can be used by developers to tackle common problems related to smart contract coding. For future work, the presented design patterns can be used to extract code building blocks, which could be integrated in automatic code generating frameworks. Further, the patterns could be incorporated into a certified set of libraries, covering typical and commonly occurring coding scenarios. Beyond that, the collated patterns could be compared to coding practices that evolve in other smart contract platforms. This could further reveal more abstract design patterns that are independent from the underlying implementation framework and are valid for smart contracts in general.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, 2nd ed. New York, NY, USA: John Wiley & Sons, Inc., 2000.

[3] M. Wöhrer and U. Zdun, "Smart contracts: Security patterns in the ethereum ecosystem and solidity," in *1st International Workshop on Blockchain Oriented Software Engineering @ SANER 2018*, March 2018. [Online]. Available: http://eprints.cs.univie.ac.at/5433/

[4] V. Garousi, M. Felderer, and M. V. Mäntylä, "The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2016.

[5] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: a critical review and guidelines," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 120–131.

[6] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including the grey literature and conducting multivocal literature reviews in software engineering," *arXiv preprint arXiv:1707.02553*, 2017.

[7] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '14. New York, NY, USA: ACM, 2014, pp. 38:1–38:10. [Online]. Available: http://doi.acm.org/10.1145/2601248.2601268

[8] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: platforms, applications, and design patterns," *arXiv preprint arXiv:1703.06322*, 2017.

[9] P. Zhang, J. White, D. C. Schmidt, and G. Lenz, "Applying software patterns to address interoperability in blockchain-based healthcare apps," *arXiv preprint arXiv:1706.03700*, 2017.

[10] A. Mavridou and A. Laszka, "Designing secure ethereum smart contracts: A finite state machine based approach," *arXiv preprint arXiv:1711.09327*, 2017.

[11] Solidity — solidity 0.4.18 documentation. [Online]. Available: https://media.readthedocs.org/pdf/solidity/develop/solidity.pdf

[12] "Ethereum contract security techniques and tips," 2017, [Online; accessed 6-September-2017]. [Online]. Available: https://github.com/ConsenSys/smart-contract-best-practices

[13] Ethereum development and dapps. [Online]. Available: https://www.reddit.com/r/ethdev/

[14] Ethereum stack exchange. [Online]. Available: https://ethereum.stackexchange.com/

[15] J. Bontje. (2015) Dapp design patterns. [Online]. Available: https://www.slideshare.net/mids106/dapp-design-patterns

[16] Smart Contracts Are Neither Smart Nor Contracts … So What Are They? [Online]. Available: https://www.infoq.com/presentations/blockchain-introduction

[17] cjgdev. (2016) Smart-contract patterns written in solidity, collated for community good. [Online]. Available: https://github.com/cjgdev/smart-contract-patterns

[18] OpenZeppelin. Openzeppelin/zeppelin-solidity: Openzeppelin, a framework to build secure smart contracts on ethereum. [Online]. Available: https://github.com/OpenZeppelin/zeppelin-solidity

[19] Modular-Network. Modular-network/ethereum-libraries: Library contracts for ethereum. [Online]. Available: https://github.com/Modular-Network/ethereum-libraries

[20] "maxwoe/solidity_patterns." [Online]. Available: https://github.com/maxwoe/solidity_patterns

[21] Oraclize - blockchain oracle service, enabling data-rich smart contracts. [Online]. Available: http://www.oraclize.it/

[22] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 270–282.

[23] M. Alharby and A. van Moorsel, "Blockchain-based smart contracts: A systematic mapping study," *arXiv preprint arXiv:1710.06372*, 2017.

[24] X. Xu, I. Weber, M. Staples, L. Zhu, J. Bosch, L. Bass, C. Pautasso, and 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.