BOZZA INFORMATIVA TESI

TITOLO:

[PLACE HOLDER]

ANALISI STATICA PER LO STUDIO DEI DESIGN PATTERN D'USO COMUNE NEGLI SMART-CONTRACT

Nel corso dello sviluppo di Solidity sono state introdotte numerose breaking changes. Le varie sorgenti da cui sono tratti i pattern design soggetti allo studio presentano incompatibilità con la più recente versione di Solidity, per questo motivo si è preferito adattare tutti gli snippet di codice al più recente Solidity 0.8.0.0.

Repository Github con codici d'esempio e parser

Ultimo aggiornamento: 10/12/2022

Changelog:

- 10/12/2022
  - Disponibile prototipo dell'utility
  - Correzioni minori del documento
- 16/11/2022
  - Inserita sezione dedicata all'analisi statica
- 07/11/2022
  - Aggiunti riferimento per design pattern
  - Aggiunti ulteriori Behavioral Pattern
  - Aggiunti Economic Pattern
  - Aggiornata bibliografica

SOMMARIO

CLASSIFICAZIONE DESIGN PATTERN PER SMART-CONTRACT

I design pattern impiegati nello sviluppo di smart-contract possono essere classificati in diverse *tipologie*, ognuna caratterizzante di uno specifico aspetto della vita del contratto:

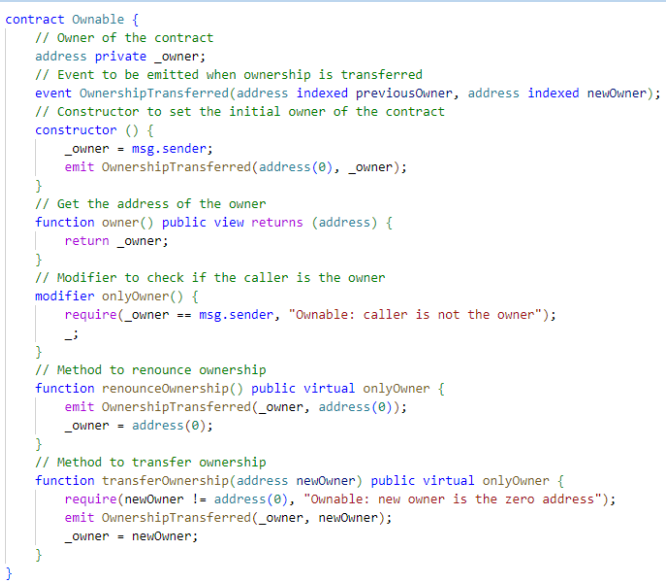
- ***Authorization***: questa tipologia di design pattern si occupa di gestire l'accesso alle funzionalità di uno smart-contract, fornendo meccanismi capaci di riservarne l'esecuzione;
- ***Behavioral****:* questa tipologia di design pattern si occupa di fornire funzionalità e meccanismi di supporto per lo svolgimento delle operazioni di uno smart-contract;
- ***Gas Economic****:* questa tipologia di design pattern si occupa di fornire i meccanismi facenti parte delle *best practices* per ridurre il consumo di *gas* durante l'esecuzione delle funzionalità dello smart-contract;
- ***Lifecycle***: questa tipologia di design pattern si occupa di gestire il processo di creazione e distruzione delle istanze di uno smart-contract;
- ***Maintenance***: questa tipologia di design pattern si occupa di fornire funzionalità utili per le operazioni di manutenzione sugli smart-contract in esecuzione;
- ***Security***: questa tipologia di design pattern si occupa di mitigare le possibili vulnerabilità di sicurezza;

AUTHORIZATION DESIGN PATTERNS

Gli *Authorization design pattern* si occupano di gestire l'accesso alle funzionalità di uno smart-contract fornendo un processo di autenticazione molto basilare. I design pattern più riconosciuti in questa tipologia sono: ***Ownership Pattern*** *e* ***Access Restriction Pattern****.*

OWNERSHIP PATTERN

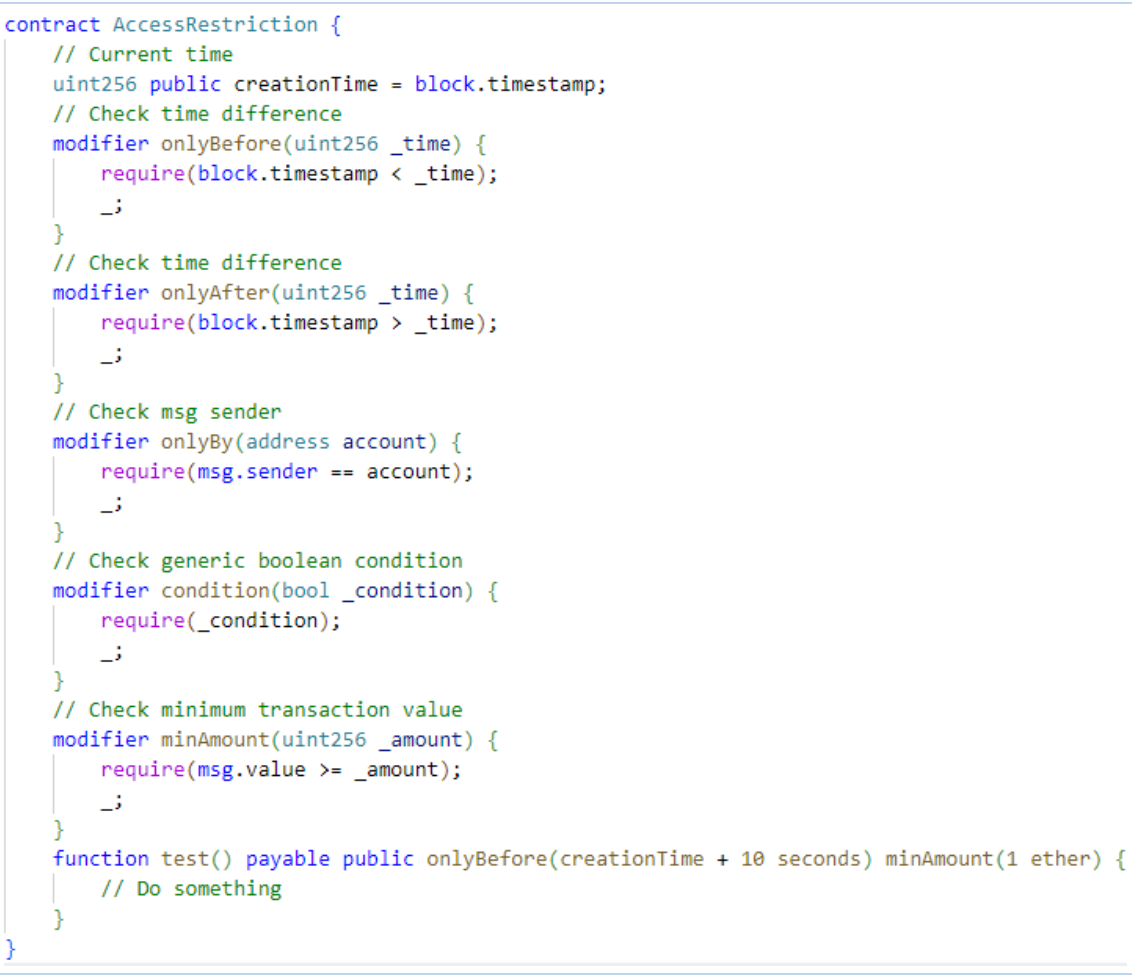
L'*Ownership pattern* ha lo scopo di riservare al creatore di uno smart-contract l'esecuzione di funzionalità critiche a cui l'utente finale non deve avere accesso. Opzionalmente possono essere presenti metodi di supporto per eseguire l'operazione di *trasferimento del proprietario*.

	Ownership Pattern
Problema	Come comportamento predefinito, ogni utente della rete Ethereum può richiamare qualunque metodo dello smart-contract ma in certi casi è necessario riservare l'esecuzione di metodi critici al solo proprietario dello smart-contract
Soluzione	Memorizzare l'indirizzo del creatore dello smart-contract e limitare l'esecuzione del metodo in base all'indirizzo del chiamante
Riconoscimento	Ricerca di un $modifier$ *con* costrutto if *o* $require$ che confronti $msg.sender$ con l'indirizzo del proprietario memorizzato
Snippet di Codice	
Riferimenti	Qui, Qui, Qui (Versione riportata)

```
contract Ownable {
    // Owner of the contract
    address private _owner;
    // Event to be emitted when ownership is transferred
    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
    // Constructor to set the initial owner of the contract
    constructor () {
        _owner = msg.sender;
        emit OwnershipTransferred(address(0), _owner);
    }
    // Get the address of the owner
    function owner() public view returns (address) {
        return _owner;
    }
    // Modifier to check if the caller is the owner
    modifier onlyOwner() {
        require(_owner == msg.sender, "Ownable: caller is not the owner");
        _;
    }
    // Method to renounce ownership
    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }
    // Method to transfer ownership
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}
```

ACCESS RESTRICTION PATTERN

L'*Access Restriction Pattern* ha lo scopo di immettere nella logica dei controlli di prerequisiti, che nel comportamento predefinito di Solidity non sono contemplati, per l'esecuzione dei metodi dello smart-contract.

	Access Restriction Pattern
Problema	Come comportamento predefinito, ogni metodo dello smart-contract viene eseguito senza il controllo di eventuali prerequisiti, ma potrebbero esserci casi d'uso in cui si vuole permettere l'esecuzione solo al raggiungimento di specifici prerequisiti
Soluzione	Definire una serie di $modifier$ generalmente applicabili che controllano i prerequisiti desiderati e applicarli nella definizione dei metodi
Riconoscimento	Ricerca di una serie $modifier$ all'interno dello smart-contract
Snippet di Codice	
Riferimento	Qui (Versione estesa), Qui, Qui

```
contract AccessRestriction {
    // Current time
    uint256 public creationTime = block.timestamp;
    // Check time difference
    modifier onlyBefore(uint256 _time) {
        require(block.timestamp < _time);
        _;
    }
    // Check time difference
    modifier onlyAfter(uint256 _time) {
        require(block.timestamp > _time);
        _;
    }
    // Check msg sender
    modifier onlyBy(address account) {
        require(msg.sender == account);
        _;
    }
    // Check generic boolean condition
    modifier condition(bool _condition) {
        require(_condition);
        _;
    }
    // Check minimum transaction value
    modifier minAmount(uint256 _amount) {
        require(msg.value >= _amount);
        _;
    }
    function test() payable public onlyBefore(creationTime + 10 seconds) minAmount(1 ether) {
        // Do something
    }
}
```

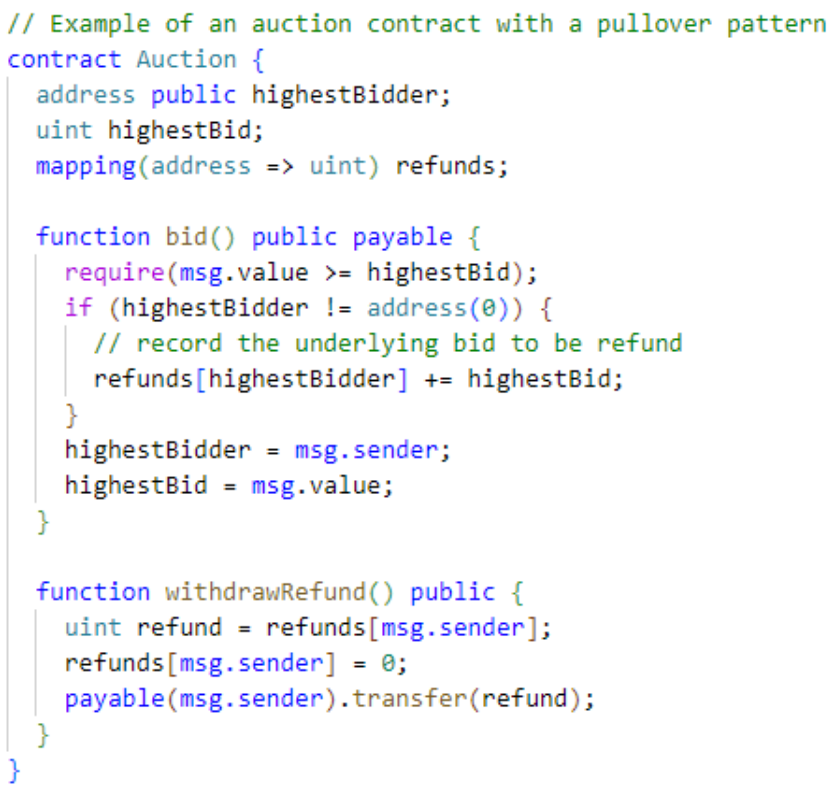
BEHAVIORAL DESIGN PATTERNS

I *Behavioral design pattern* si occupano di fornire funzionalità e meccanismi di supporto per lo svolgimento delle operazioni di uno smart-contract.

I design pattern più riconosciuti in questa tipologia sono: ***Pull Payment (Pull over Push) Pattern***, ***State Machine Pattern***, ***Oracle Pattern***, ***Commit & Reveal Pattern***, ***Randomness Pattern*** *e* ***Guard Check Pattern***.

PULL PAYMENT *(PULL OVER PUSH)* PATTERN

Il *Full Payment Pattern*, spesso chiamato anche *Pull Over Push Pattern*, ha lo scopo di fornire un meccanismo per eseguire pagamenti (trasferimenti di criptovaluta) in modo sicuro.

	Full Payment *(Pull over Push)* Pattern
Problema	L'invio di ether comporta una chiamata all'entità ricevente. Ci sono diversi motivi per cui questa chiamata esterna potrebbe fallire: se l'indirizzo ricevente è un contratto, potrebbe essere implementata una funzione di $fallback()$ che lancia semplicemente un'eccezione, una volta chiamata. Un altro motivo di fallimento è l'esaurimento del "gas".
Soluzione	Seguire un semplice principio: non fidarsi mai che le chiamate esterne vengano eseguite senza lanciare un errore. Si può dire che è responsabilità del destinatario assicurarsi di essere in grado di ricevere il denaro. Per questo motivo la situazione ideale è far sì che sia il destinatario a innescare il trasferimento
Riconoscimento	Ricerca del trasferimento ad opera del $sender$: $msg.sender.transfer()$
Snippet di Codice	
Riferimento	Qui, Qui

```
// Example of an auction contract with a pullover pattern
contract Auction {
  address public highestBidder;
  uint highestBid;
  mapping(address => uint) refunds;

  function bid() public payable {
    require(msg.value >= highestBid);
    if (highestBidder != address(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;
    payable(msg.sender).transfer(refund);
  }
}
```

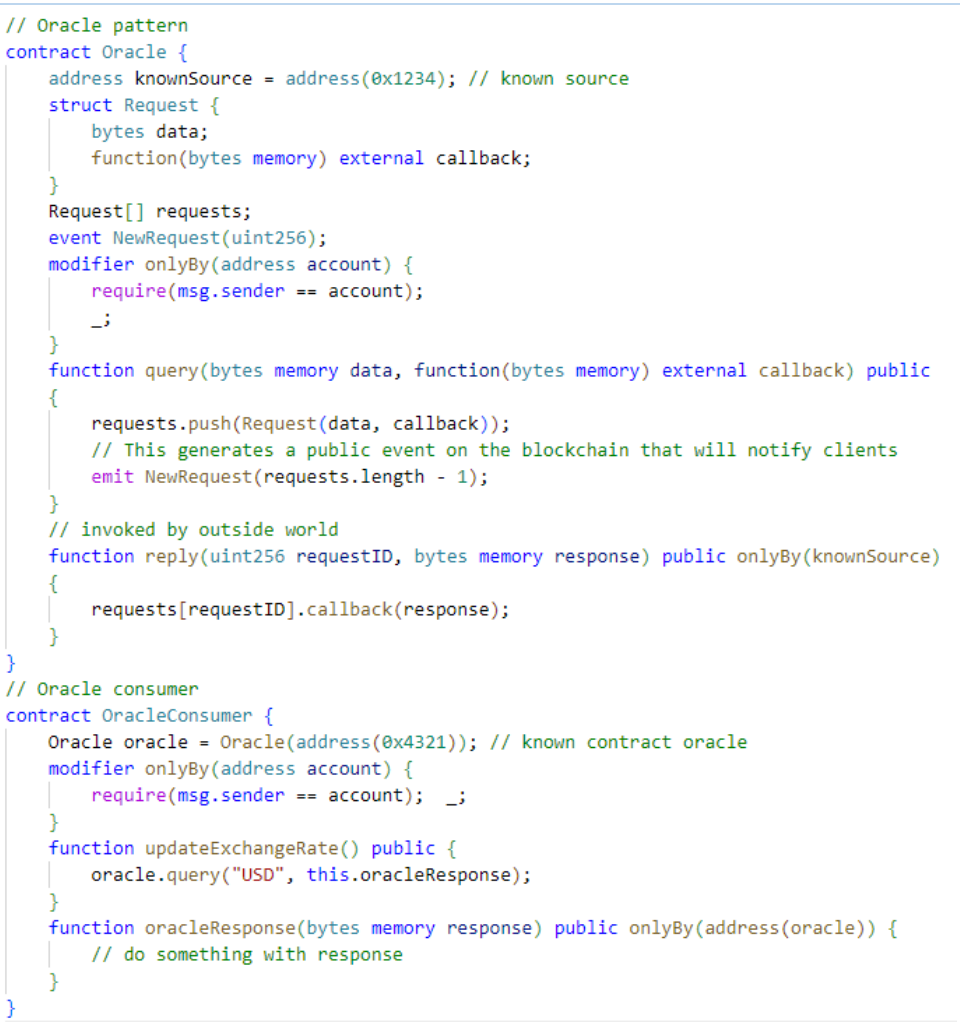
STATE MACHINE PATTERN

Lo *State Machine Pattern* ha lo scopo di fornire un meccanismo per consentire a uno smart-contract di passare, durante la sua esecuzione, attraverso diverse fasi con diverse funzionalità corrispondenti esposte.

	State Machine Pattern
Problema	Consideriamo un contratto che deve passare da uno stato iniziale, attraverso diversi stati intermedi, allo stato finale nel corso della sua attività. In ognuno di questi stati il contratto deve comportarsi in modo diverso e fornire diverse funzionalità ai suoi utenti. Il comportamento descritto può essere osservato in una moltitudine di casi d'uso: aste, gioco d'azzardo, crowdfunding e ect.
Soluzione	Implementare una macchina a stati per modellare e rappresentare le diverse fasi del contratto comportamentale e le loro transizioni.
Riconoscimento	Ricerca di un $enum$ che contenga gli stati e il $modifier$ che controlla lo stato
Snippet di Codice	`// Example of an deposit contract with a state machine pattern` `contract DepositLock {` `  enum Stages {` `  AcceptingDeposits,` `  FreezingDeposits,` `  ReleasingDeposits` `  }` `  Stages public stage = Stages.AcceptingDeposits;` `  uint256 public creationTime = block.timestamp;` `  mapping(address => uint256) balances;` `  modifier atStage(Stages _stage) {` `  require(stage == _stage);` `  _;` `  }` `  modifier timedTransitions() {` `  if (stage == Stages.AcceptingDeposits && block.timestamp >= creationTime + 1 days) nextStage();` `  if (stage == Stages.FreezingDeposits && block.timestamp >= creationTime + 8 days) nextStage();` `  _;` `  }` `  function nextStage() internal {` `  stage = Stages(uint256(stage) + 1);` `  }` `  function deposit() public payable timedTransitions atStage(Stages.AcceptingDeposits) {` `  balances[msg.sender] += msg.value;` `  }` `  function withdraw() public timedTransitions atStage(Stages.ReleasingDeposits) {` `  uint256 amount = balances[msg.sender];` `  balances[msg.sender] = 0;` `  payable(msg.sender).transfer(amount);` `  }` `}`
Riferimento	Qui

ORACLE PATTERN

L'*Oracle Pattern* ha lo scopo di fornire ad uno smart-contract un meccanismo per ottenere delle informazioni dal mondo esterno, necessarie al corretto funzionamento del contratto e non presenti all'interno della rete Ethereum.

	Oracle Pattern
Problema	Si immagini un caso d'uso in cui lo smart-contract necessiti di informazioni presenti al di fuori della blockchain. Gli smart-contract Ethereum non possono acquisirle direttamente poiché non possono contattare il mondo esterno, al contrario si affidano al mondo esterno che spinge le informazioni nella rete
Soluzione	Richiedere i dati esterni attraverso un servizio *oracolo* che è collegato al mondo esterno e funge da provider di dati
Riconoscimento	Ricerca di un $oracolo$ ovvero di uno smart-contract contenente una $struct$ per le richieste e i metodi $query\ e\ reply$
Snippet di Codice	
Riferimento	Qui, Qui (Versione basata su Oraclize)

```
// Oracle pattern
contract Oracle {
    address knownSource = address(0x1234); // known source
    struct Request {
        bytes data;
        function(bytes memory) external callback;
    }
    Request[] requests;
    event NewRequest(uint256);
    modifier onlyBy(address account) {
        require(msg.sender == account);
        _;
    }
    function query(bytes memory data, function(bytes memory) external callback) public
    {
        requests.push(Request(data, callback));
        // This generates a public event on the blockchain that will notify clients
        emit NewRequest(requests.length - 1);
    }
    // invoked by outside world
    function reply(uint256 requestID, bytes memory response) public onlyBy(knownSource)
    {
        requests[requestID].callback(response);
    }
}
// Oracle consumer
contract OracleConsumer {
    Oracle oracle = Oracle(address(0x4321)); // known contract oracle
    modifier onlyBy(address account) {
        require(msg.sender == account);  _;
    }
    function updateExchangeRate() public {
        oracle.query("USD", this.oracleResponse);
    }
    function oracleResponse(bytes memory response) public onlyBy(address(oracle)) {
        // do something with response
    }
}
```

COMMIT & REVEAL PATTERN

Il *Commit & Reveal Pattern* ha lo scopo di fornire un meccanismo per consentire a uno smart-contract di passare attraverso diverse fasi con funzionalità corrispondenti esposte.

<table>
<tr><th></th><th>Commit & Reveal Pattern</th></tr>
<tr><td>Problema</td><td>In una blockchain, tutti i dati e tutte le transazioni sono visibili pubblicamente. Si può facilmente immaginare un caso d’uso in cui le interazioni con lo smart-contract, particolarmente i valori dei parametri inviati, debbano essere trattate in modo confidenziale</td></tr>
<tr><td>Soluzione</td><td>Implementare uno schema di commitment. Uno schema di commitment è un algoritmo crittografico utilizzato per consentire a qualcuno di impegnarsi su un valore tenendolo nascosto ad altri con la possibilità di rivelarlo in seguito. I valori in uno schema di commitment sono vincolanti, ovvero nessuno può cambiarli una volta impegnati. Lo schema prevede due fasi: una fase di commit in cui viene scelto e specificato un valore e una fase di reveal in cui il valore viene rivelato e controllato</td></tr>
<tr><td>Riconoscimento</td><td>Ricerca dei metodi $commit\ e\ reveal$, dei log $LogCommit\ e\ LogReveal$ e relativi $emit$, $struct$ dei commit</td></tr>
<tr><td>Snippet di codice</td><td>
<pre>
// Commit-reveal pattern
contract CommitReveal {
    // Commit structure
    struct Commit {string choice; string secret; string status;}
    // Commit mapping
    mapping(address => mapping(bytes32 => Commit)) public userCommits;
    event LogCommit(bytes32, address);
    event LogReveal(bytes32, address, string, string);
    // Commit function to initiate a commit
    function commit(bytes32 _commit) public returns (bool success) {
        Commit storage userCommit = userCommits[msg.sender][_commit];
        if (bytes(userCommit.status).length != 0) {
            return false; // commit has been used before
        }
        userCommit.status = "c"; // comitted
        emit LogCommit(_commit, msg.sender);
        return true;
    }
    function reveal(string calldata _choice, string calldata _secret, bytes32 _commit) public returns (bool success) {
        Commit storage 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(abi.encodePacked(_choice, _secret))) {
            return false; // hash does not match commit
        }
        userCommit.choice = _choice;
        userCommit.secret = _secret;
        userCommit.status = "r"; // revealed
        emit LogReveal(_commit, msg.sender, _choice, _secret);
        return true;
    }
    function traceCommit(address _address, bytes32 _commit) public view
        returns (string memory choice, string memory secret, string memory status) {
        Commit storage userCommit = userCommits[_address][_commit];
        // Check if commit is revealed
        require(bytes(userCommit.status)[0] == "r");
        return (userCommit.choice, userCommit.secret, userCommit.status);
    }
}
</pre>
</td></tr>
<tr><td>Riferimento</td><td>Qui</td></tr>
</table>

RANDOMNESS PATTERN

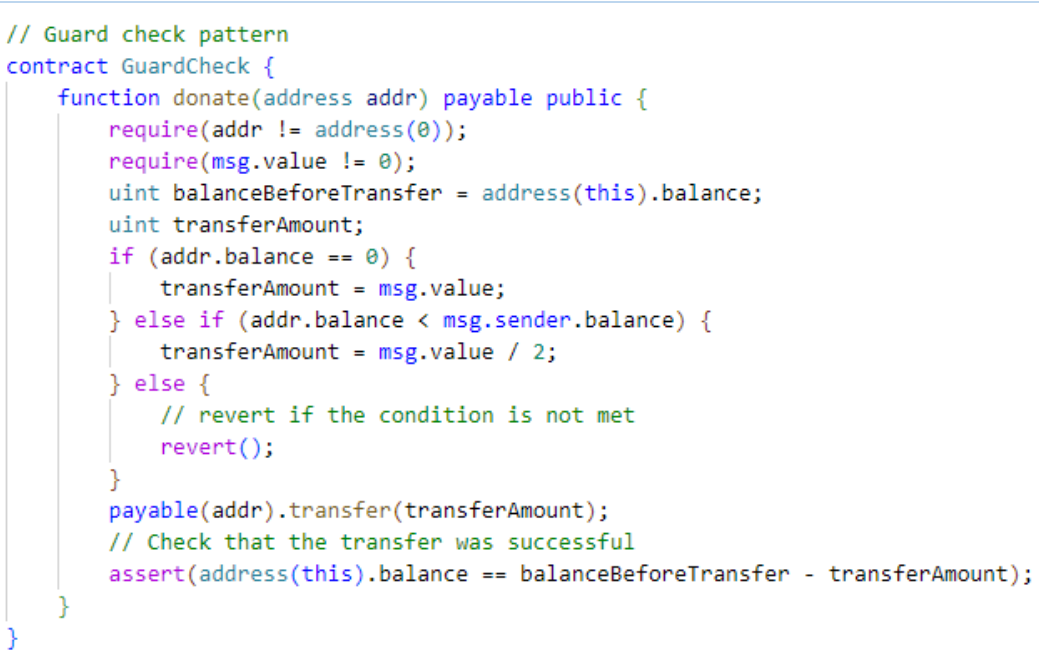
Il *Randomness Pattern* ha lo scopo di fornire un meccanismo per consentire a uno smart-contract di generare un numero casuale, appartenente a un intervallo predefinito, in un ambiente deterministico come la blockchain.

	Randomness Pattern
Problema	La casualità nei sistemi informatici, e soprattutto in Ethereum, è notoriamente difficile da ottenere. Per quanto riguarda Ethereum, la rete è una macchina di Turing deterministica, senza alcuna casualità intrinseca ma, nonostante ciò, la necessità di casualità è assai elevata. Un'altra problematica per la casualità è la natura pubblica di una blockchain. Lo stato interno di un contratto, così come l'intera storia di una blockchain, è visibile al pubblico. Pertanto, è difficile trovare una fonte sicura di entropia. Un caso d'uso molto diffuso è lo smart-contract che fornisce giochi, i quali si basano su un fattore di casualità per determinare il vincitore.
Soluzione	Sono state ideate diverse soluzioni per superare queste limitazioni, ciascuna con i rispettivi vantaggi e svantaggi: • *Block hash PRNG*: l'hash del blocco viene usato come sorgente di casualità; • *Oracle RNG:* si usa un oracolo il quale fa da provider per la casualità; • *Collaborative PRNG*: una generazione collaborativa di casualità dentro la blockchain;
Riconoscimento	• Per individuare l'implementazione più semplice di randomness basta individuare l'utilizzo delle seguenti istruzioni: ➢ $uint(blockhash(block.number\ -\ 1))$, da non utilizzare per scommesse in quanto è facilmente determinabile; ➢ $uint(keccak256(abi.encodePacked(blockhash(block.number-1), seed)))$, in cui il $seed$ viene generato dall'utente e fornito come parametro; ➢ $uint(keccak256($ $abi.encodePacked(block.timestamp, msg.sender, block.difficulty)))$ • Le altre implementazioni sono ostiche da analizzare
Snippet di codice	`contract SimpleRandom {` `  // Simple random number generator` `  function simpleRandomNumber() internal view returns (uint) {` `  return uint(blockhash(block.number - 1));` `  }` `  // Random number generator based on user's seed` `  function seededRandomNumber(string calldata seed) internal view returns (uint) {` `  return uint(keccak256(abi.encodePacked(blockhash(block.number-1), seed)));` `  }` `  // Generate random based on time, address and block difficulty` `  function random() public view returns (uint) {` `  return uint(keccak256(abi.encodePacked(block.timestamp, msg.sender, block.difficulty)));` `  }` `}`
Riferimento	Qui (Spiegazione e versione estesa)

GUARD CHECK PATTERN

Il *Guard Check Pattern* ha lo scopo di fornire un meccanismo al creatore di uno smart-contract per assicurarsi che il comportamento dello smart-contract sia quello previsto e che i parametri di input siano validi.

Potremmo immaginare questo design pattern come una implementazione di numerosi test eseguiti a runtime per valutare lo stato d'esecuzione.

	Guard Check Pattern
Problema	In una blockchain non vi sono regolatori o mediatori ma vi è bisogno di protezioni o controlli per assicurare che la logica degli smart-contract funzioni come specificato. Uno smart-contract dovrebbe verificare tutti i prerequisiti della funzionalità e procedere solo se tutto è come previsto. In caso di problemi, il contratto dovrebbe ripristinare tutte le modifiche apportate al suo stato.
Soluzione	Per raggiungere questi obbiettivi, Solidity sfrutta il modo in cui l'EVM gestisce gli errori: per mantenere l'atomicità, tutte le modifiche vengono annullate e l'intera transazione non ha effetto. Solidity utilizza le eccezioni per innescare questi errori e ripristinare lo stato. Solidity offre diversi modi per attivare tali eccezioni.
Riconoscimento	Per riconoscere tale pattern si ricerca l'utilizzo di: • $assert(condition)$: usato solo per verificare la presenza di errori interni e per controllare gli invarianti (asserzioni sempre $true$); ➢ Rimborsa tutto il gas che non è stato consumato nel momento in cui viene lanciata l'eccezione; • $require(condition)$: usato solo per garantire condizioni valide, come gli input o le variabili di stato del contratto, o per convalidare i valori di ritorno da chiamate a contratti esterni; ➢ Consuma tutto il gas incluso nella transazione • $revert()$: usato per ripristinare le modifiche apportate allo stato, viene normalmente lanciato in caso di fallimento da $require\ o\ assert$ ma potrebbe essere utilizzato all'interno di un controllo fatto da un costrutto if.
Snippet di codice	
Riferimento	Qui (Spiegazione)

GAS ECONOMIC PATTERNS

I *Gas Economic Patterns* si occupano di fornire i meccanismi facenti parte delle *best practices* per ridurre il consumo di gas durante l'esecuzione delle funzionalità dello smart-contract.

Operazioni con consumi di gas altamente variabili e imprevedibili sono un problema per gli smart-contract, in quanto comportano il rischio che le transazioni rimangano senza gas e portino a comportamenti indesiderati. Pertanto, è auspicabile un fabbisogno di gas basso, stabile e prevedibile.

I design pattern più riconosciuti in questa tipologia sono**: *String Equality Comparison Pattern*, *Tight Variable Packing Pattern*** *e* ***Memory Array Building Pattern***.

STRING EQUALITY COMPARISON PATTERN

Lo *String Equality Comparison Pattern* ha lo scopo di fornire un meccanismo per poter verificare l'uguaglianza di due stringhe in un modo che minimizza il consumo medio del gas per un gran numero di input diversi.

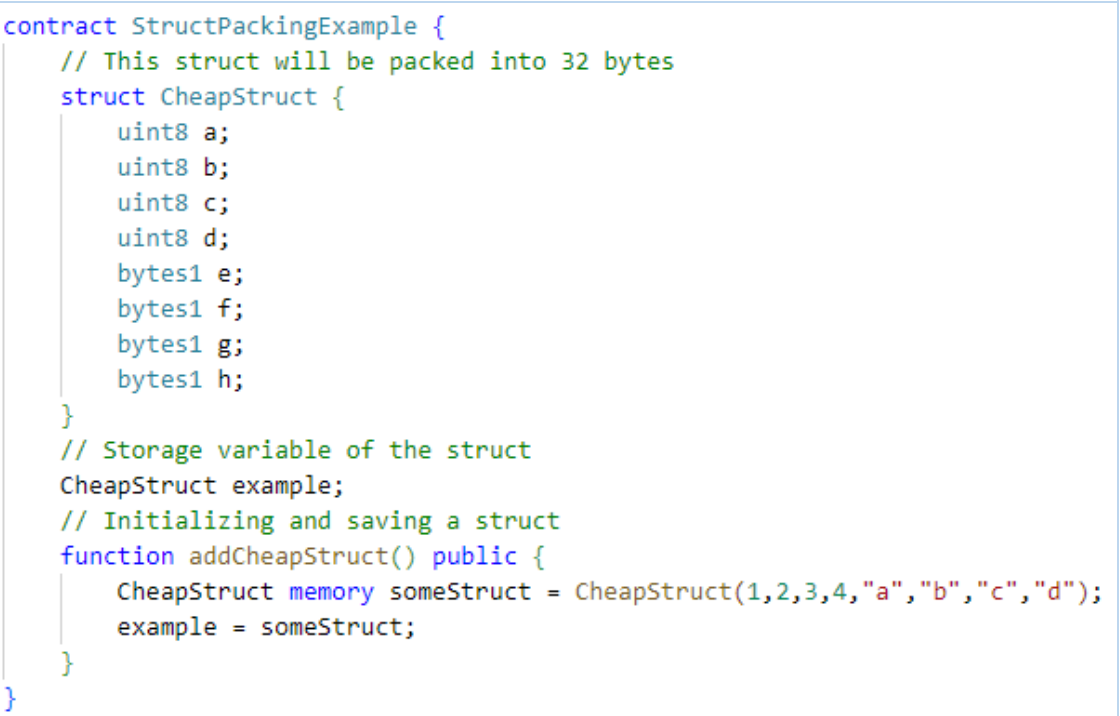
	String Equality Comparison Pattern
Problema	Confrontare le stringhe in altri linguaggi di programmazione è un compito banale, vi sono metodi o pacchetti integrati che possono verificare l'uguaglianza di due input con una sola chiamata, ad esempio String1.equals(String2) in Java. Solidity, nella sua versione 0.8.0.0, non supporta alcuna funzionalità di questo tipo al momento della stesura del presente documento.
Soluzione	La soluzione proposta è basata sull'utilizzo di una funzione di hash per il confronto, opzionalmente combinata con un controllo della corrispondenza della lunghezza delle stringhe fornite, per eliminare fin dall'inizio le coppie con lunghezze diverse.
Riconoscimento	Per riconoscere tale pattern si ricerca l'utilizzo di una condizione booleana di confronto fra i risultati della funzione $keccak256()$
Snippet di codice	`contract SimpleContractExample {` `  function hashCompareCheck(string memory _a, string memory _b) internal pure returns (bool) {` `  return` `  keccak256(abi.encodePacked(_a)) == keccak256(abi.encodePacked(_b));` `  }` `  // In cases of different length, the function will use less gas than the one above` `  function hashCompareWithLengthCheck(string memory _a, string memory _b) internal pure returns (bool) {` `  if (bytes(_a).length != bytes(_b).length) {` `  return false;` `  } else {` `  return keccak256(abi.encodePacked(_a)) == keccak256(abi.encodePacked(_b));` `  }` `  }` `}`
Riferimento	Qui (Spiegazione)

TIGHT VARIABLE PACKING PATTERN

Il *Tight Variable Packing Pattern* ha lo scopo di fornire un meccanismo per ottimizzare il consumo di gas quando si memorizzano o si caricano variabili di dimensioni statiche.

Lo storage in Ethereum è una struttura chiave-valore con chiavi e valori di 32 byte ciascuno. Quando lo storage viene allocato, tutte le variabili di dimensioni statiche, eccetto le mappature e gli array di dimensioni dinamiche, vengono scritte una dopo l'altra, nell'ordine in cui sono state dichiarate, a partire dalla posizione 0. I tipi di dati più comunemente utilizzati come $bytes32$, $uint$, int occupano esattamente uno slot da 32 byte nello storage perciò possono essere letti o scritti in un'unica operazione. Sarebbe ottimale organizzare, ovvero ordinare la dichiarazione, dei tipi più piccoli come $bytes16$ $uint8$, e così via, in modo che l'EVM possa raggrupparli in un singolo slot da 32 byte, utilizzando così meno memoria e pure risparmiando gas poichè l'EVM combinerà più letture o scritture in un'unica operazione.

Il *packing* delle variabili viene spesso usato per salvare spazio e gas per raggruppare risultati booleani, i quali necessità di un solo bit, oppure per raggruppare in 32byte una composizione di *unsigned int*.

	Tight Variable Packing Pattern
Problema	Salvare o caricare dati di dimensione inferiori a 32 byte può causare uno spreco di gas correlato all'esecuzione di operazioni di lettura e scrittura superflue su uno storage basato a slot di 32 byte
Soluzione	Organizzare, ovvero ordinare la dichiarazione, dei tipi più piccoli come bytes16 uint8, e così via, in modo che l'EVM possa raggrupparli in un singolo slot da 32 byte
Riconoscimento	Per riconoscere tale pattern si ricerca la definizione di uno $struct$ contenente tipi di dati la cui somma dello spazio occupato sia minore o uguale, quando possibile, a 32 $byte$
Snippet di codice	
Riferimento	Qui (Spiegazione), , Qui (Articolo IEEE Tab. Packing Variables)

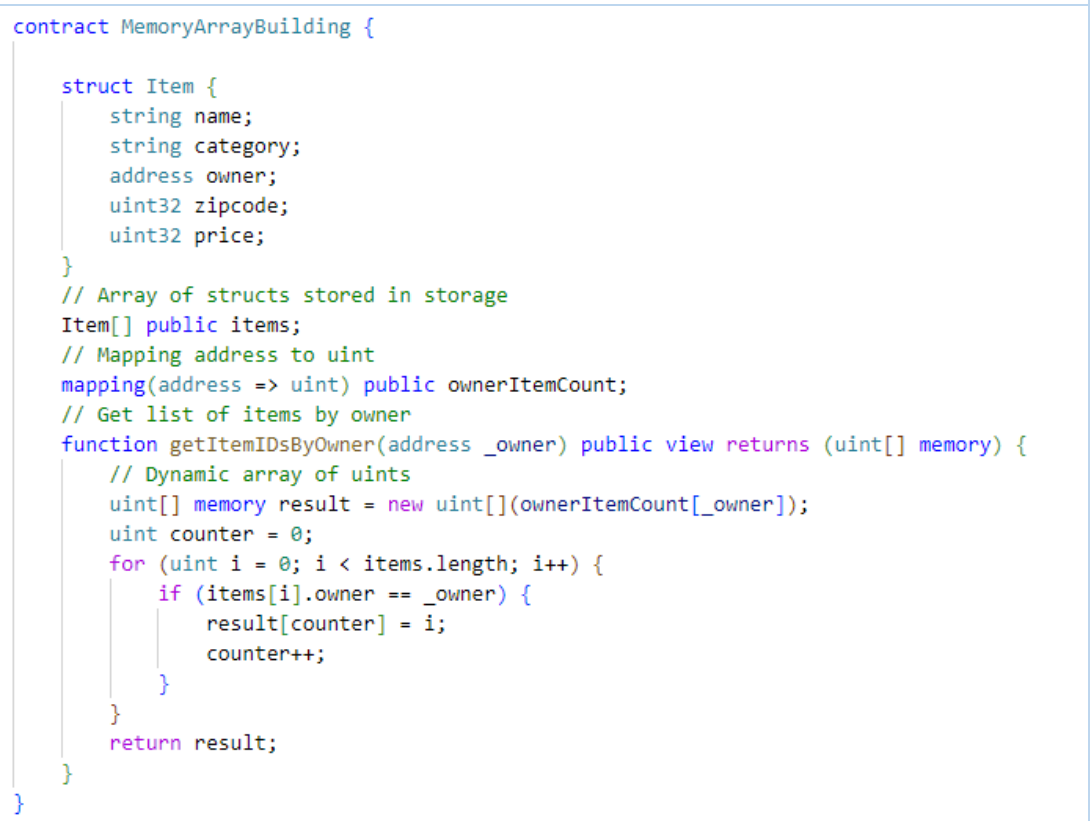
```
contract StructPackingExample {
    // This struct will be packed into 32 bytes
    struct CheapStruct {
        uint8 a;
        uint8 b;
        uint8 c;
        uint8 d;
        bytes1 e;
        bytes1 f;
        bytes1 g;
        bytes1 h;
    }
    // Storage variable of the struct
    CheapStruct example;
    // Initializing and saving a struct
    function addCheapStruct() public {
        CheapStruct memory someStruct = CheapStruct(1,2,3,4,"a","b","c","d");
        example = someStruct;
    }
}
```

MEMORY ARRAY BUILDING PATTERN

Il *Memory array Building Pattern* ha lo scopo di fornire un meccanismo per aggregare e recuperare i dati dallo storage di uno smart-contract in modo efficiente dal punto di vista del gas.

L'interazione con lo storage di uno smart-contract sulla blockchain è una delle operazioni più costose dell'EVM. Pertanto, è necessario memorizzare solo i dati necessari ed evitare, se possibile, la ridondanza. Nei sistemi tradizionali l'unico costo rilevante delle query in uno storage è il tempo, mentre in Ethereum anche semplici interrogazioni possono costare una quantità sostanziale di gas, che ha un riscontro monetario diretto.

Un modo per mitigare i costi del gas è dichiarare una variabile pubblica, il che comporta la creazione di un getter in background che permette di accedere gratuitamente al valore della variabile.

	Memory Array Building Pattern
Problema	Si immagini un caso d'uso in cui si voglia aggregare dati provenienti da diverse fonti, ciò richiederebbe una grande quantità di letture dallo storage e sarebbe quindi particolarmente costoso
Soluzione	Sfruttare il $modifier\ view$ di Solidity, che consente di aggregare e leggere i dati dallo storage del contratto senza alcun costo associato. Ogni volta che viene richiesto un lookup, viene ricostruito un array in $memory$, invece di salvarlo nello $storage$. Nei sistemi convenzionali questa operazione risulterebbe assai inefficiente, ma in Solidity la soluzione proposta è più efficiente perché le funzioni dichiarate con il $modifier\ view$ non sono autorizzate a scrivere sullo $storage$ e quindi non modificano lo stato della blockchain. Poiché lo stato della blockchain rimane invariato, non è necessario trasmettere una transazione alla rete. Nessuna transazione significa nessun consumo di gas, il che rende gratuita la chiamata della funzione, purché sia chiamata esternamente e non da un altro contratto. In tal caso, sarebbe necessaria una transazione e verrebbe consumato del gas.
Riconoscimento	Per riconoscere tale pattern si ricerca la definizione di una funzione con $modifier\ view$ che ritorna un tipo $memory$
Snippet di codice	
Riferimento	Qui (Spiegazione), Qui (Articolo IEEE Tab. Limit Storage)

```
contract MemoryArrayBuilding {

    struct Item {
        string name;
        string category;
        address owner;
        uint32 zipcode;
        uint32 price;
    }
    // Array of structs stored in storage
    Item[] public items;
    // Mapping address to uint
    mapping(address => uint) public ownerItemCount;
    // Get list of items by owner
    function getItemIDsByOwner(address _owner) public view returns (uint[] memory) {
        // Dynamic array of uints
        uint[] memory result = new uint[](ownerItemCount[_owner]);
        uint counter = 0;
        for (uint i = 0; i < items.length; i++) {
            if (items[i].owner == _owner) {
                result[counter] = i;
                counter++;
            }
        }
        return result;
    }
}
```

LIFECYCLE DESIGN PATTERNS

I *lifecycle design pattern* si occupano di fornire meccanismi per la creazione e la distruzione delle istanze degli smart-contract. I design pattern più riconosciuti in questa tipologia sono: ***Mortal Pattern*** *e* ***Auto Deprecation Pattern****.*

MORTAL PATTERN

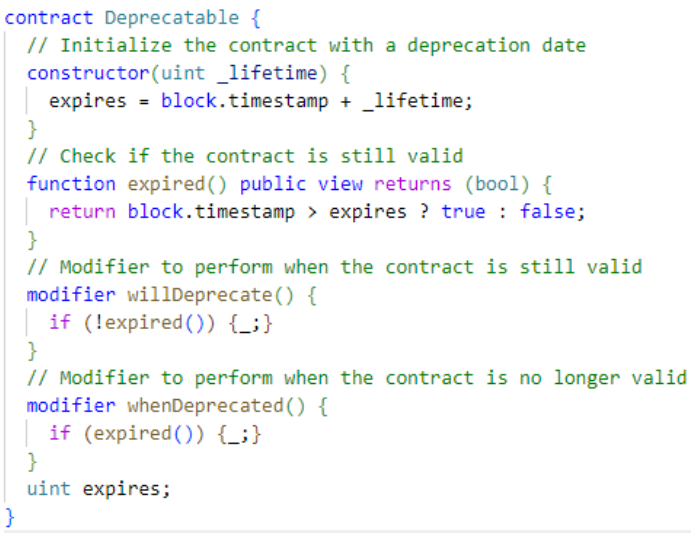
Questo pattern ha lo scopo di fornire al creatore dello smart-contract un metodo per terminarne l'esecuzione e distruggerlo.

Poiché è necessario riconoscere il creatore dello smart-contract, l'accesso a tale metodo viene spesso gestito attraverso un *ownership* pattern design.

	Mortal Pattern
Problema	L'esistenza di uno smart-contract in attività è legata all'esistenza della rete Ethereum. Se il periodo di attività dello smart-contract è terminato, allora deve essere possibile interromperne l'esecuzione e distruggerlo
Soluzione	Utilizzare una chiamata $selfdestruct([address])$ all'interno di un metodo che esegue un controllo preliminare dell'autorizzazione del richiedente
Note	• Per smart-contract di vecchia data anziché $selfdestruct([address])$ è possibile trovare $suicide([address])$. ➢ Riferimento: EIP6 • Sempre per vecchie versioni, la funzione canonica di terminazione prende il nome di $kill()$ anziché $destroy()$
Riconoscimento	Ricerca di un costrutto $if\ o\ require$ o un $modifier$ per l'autenticazione seguito dalla chiamata $selfdestruct([address])$
Snippet di Codice	`// Import ownership pattern` `import "../Authorization/ownership_pattern.sol";` `contract Mortal is Ownable {` `  // Make use of the ownership modifier` `  function destroy() public onlyOwner {` `  // If the owner calls this function, the funds are sent to the owner and the contract is destroyed` `  selfdestruct(payable(owner()));` `  }` `  function destroyWithoutModifier() public {` `  // If the owner calls this function, the funds are sent to the owner and the contract is destroyed` `  if (msg.sender == owner())` `  selfdestruct(payable(owner()));` `  }` `}`
Riferimento	Qui, Qui,

AUTO DEPRECATION PATTERN

Questo pattern ha lo scopo di fornire un meccanismo automatico che allo scadere di un definito quanto di tempo interrompa l'esecuzione della richiesta da parte dello smart-contract.

	Auto Deprecation Pattern
Problema	Si ipotizza uno scenario d'uso in cui dopo un quanto di tempo lo smart-contract debba terminare l'esecuzione della richiesta, ovvero si verifica un timeout
Soluzione	Definire un quanto di tempo e applicare modificatori nelle definizioni di funzione per disabilitarne l'esecuzione se la data di scadenza è stata raggiunta.
Note	• Da Solidity 0.7.0 la *keyword now* è stata deprecata in favore di $block.timestamp$;
Riconoscimento	Ricerca della funzione $expired()$ e $modifier$ che fanno uso di tale funzione
Snippet di Codice	
Riferimento	Qui, Qui, Qui

MAINTENANCE DESIGN PATTERNS

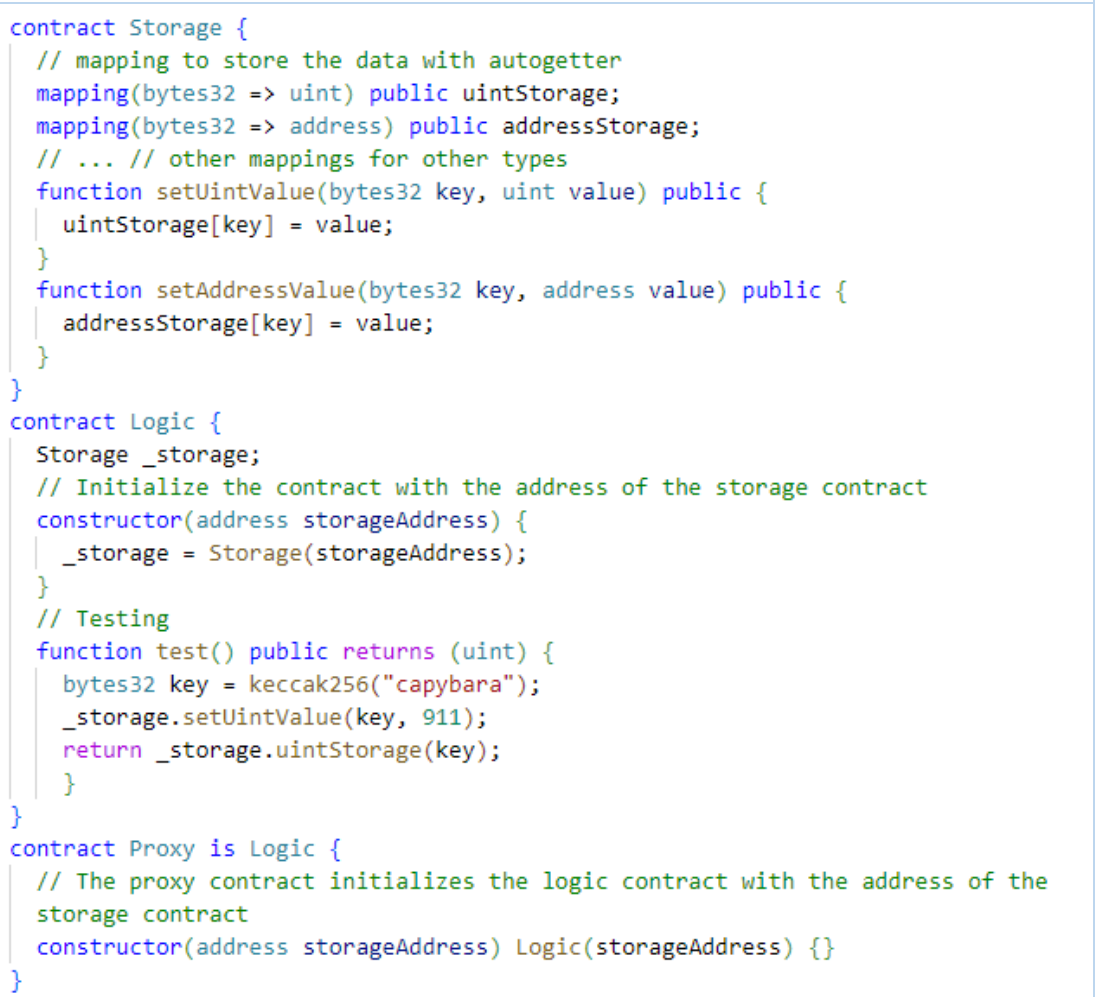
I m*aintenance design pattern* si occupano di fornire funzionalità agli smart-contract in esecuzione, definendo modalità di struttura del codice o meccanismi interni.

I design pattern più riconosciuti in questa tipologia sono: ***Data Segregation (Eternal Storage) Pattern, Relay (Proxy Delegate) Pattern, Satellite Pattern e Register Pattern***.

DATA SEGREGATION (ETERNAL STORAGE) PATTERN

Il *Data Segregation Pattern,* spesso chiamato anche *Eternal Storage Pattern*, ha lo scopo di segmentare la logica di uno smart-contract e i suoi dati per evitare costose migrazioni di dati nel momento in cui uno smart-contract viene aggiornato.

Opzionalmente, per avere un disaccoppiamento ottimale è necessario usare uno smart-contract *proxy* che si interfaccia con l'utente. Il proxy farà uso della più recente implementazione della logica dello smart-contract desiderato che a sua volta potrà comunicare con lo smart-contract dei dati.

	Data Segregation (Eternal Storage) Pattern
Problema	I dati di uno smart-contract e la sua logica sono di solito mantenuti nello stesso smart-contract, il che porta a un accoppiamento stretto. Quando uno smart-contract viene aggiornato, i dati della versione precedente devono essere migrati alla nuova versione dello smart-contract
Soluzione	Disaccoppiare i dati dalla logica operativa in smart-contract separati
Riconoscimento	Ricerca di un contratto $Storage$ che esponga eventuali $getter\ e\ setter$. Possibile implementazione del $Proxy$.
Snippet di Codice	
Riferimento	Qui, Qui, Qui, Qui (Versione estesa dello storage), Qui (Articolo IEEE Tab. Data Contract)

```
contract Storage {
  // mapping to store the data with autogetter
  mapping(bytes32 => uint) public uintStorage;
  mapping(bytes32 => address) public addressStorage;
  // ... // other mappings for other types
  function setUintValue(bytes32 key, uint value) public {
    uintStorage[key] = value;
  }
  function setAddressValue(bytes32 key, address value) public {
    addressStorage[key] = value;
  }
}
contract Logic {
  Storage _storage;
  // Initialize the contract with the address of the storage contract
  constructor(address storageAddress) {
    _storage = Storage(storageAddress);
  }
  // Testing
  function test() public returns (uint) {
    bytes32 key = keccak256("capybara");
    _storage.setUintValue(key, 911);
    return _storage.uintStorage(key);
    }
}
contract Proxy is Logic {
  // The proxy contract initializes the logic contract with the address of the
  storage contract
  constructor(address storageAddress) Logic(storageAddress) {}
}
```

RELAY (PROXY DELEGATE) PATTERN

Il *Relay Pattern,* spesso chiamato anche *Proxy Delegate Pattern*, ha lo scopo di fornire al creatore di uno smart-contract un meccanismo per aggiornare il proprio smart-contract senza invalidare l'address attualmente utilizzato, permettendo così un disaccoppiamento fra il canale di comunicazione con l'utente e la logica dello smart-contract.

Poiché è necessario riconoscere il creatore dello smart-contract, l'accesso al metodo viene spesso gestito attraverso un *ownership* pattern design.

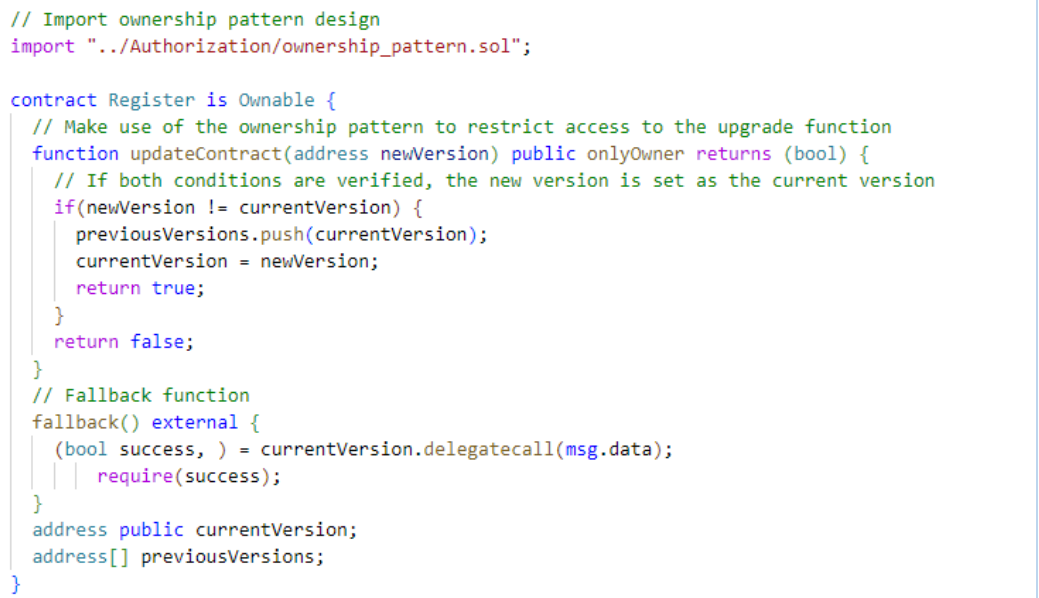
	Relay (Proxy Delegate) Pattern
Problema	Gli utenti dello smart-contract devono interfacciarsi sempre con l'ultima versione disponibile senza dover effettuare alcuna azione
Soluzione	Gli utenti si interfacciano con uno smart-contract *proxy* che inoltrerà tutte le richieste alla più recente versione dello smart-contract desiderato
Note	• In versioni datate di Solidity, la funzione $fallback()$ potrebbe essere definita come $function();$
Riconoscimento	Ricerca di un metodo protetto da autenticazione che aggiorni la variabile che punta all'indirizzo dello smart-contract in esecuzione e una funzione generica $fallback()$
Snippet di Codice	`// Import ownership pattern design` `import "../Authorization/ownership_pattern.sol";` `contract Relay is Ownable {` `  // Initialize the relay with current version of the logic contract` `  constructor(address initAddr) {` `  currentVersion = initAddr;` `  }` `  // Make use of the ownership pattern to restrict access to the upgrade function` `  function updateContract(address newVersion) public onlyOwner {` `  // If the condition is verified, the new version is set as the current version` `  currentVersion = newVersion;` `  }` `  // Fallback function` `  fallback() external {` `  (bool success, ) = currentVersion.delegatecall(msg.data);` `  require(success);` `  }` `  address public currentVersion;` `}`
Riferimento	Qui, Qui, Qui (Versione estesa), Qui (Versione ottimizzata per gas), , Qui (Articolo IEEE Tab. Proxy)

REGISTER PATTERN

Il *Register Pattern* ha lo scopo di fornire al creatore di uno smart-contract un meccanismo per aggiornare il proprio smart-contract attraverso un *registro* che possa fornire su richiesta agli utenti l'address dell'ultima versione disponibile dello smart-contract.

Poiché è necessario riconoscere il creatore dello smart-contract, l'accesso al metodo viene spesso gestito attraverso un *ownership* pattern design.

Il *Relay Pattern* rende pressocché obsoleto questo pattern, ma è probabile che vecchi smart-contract possano ancora utilizzarlo.

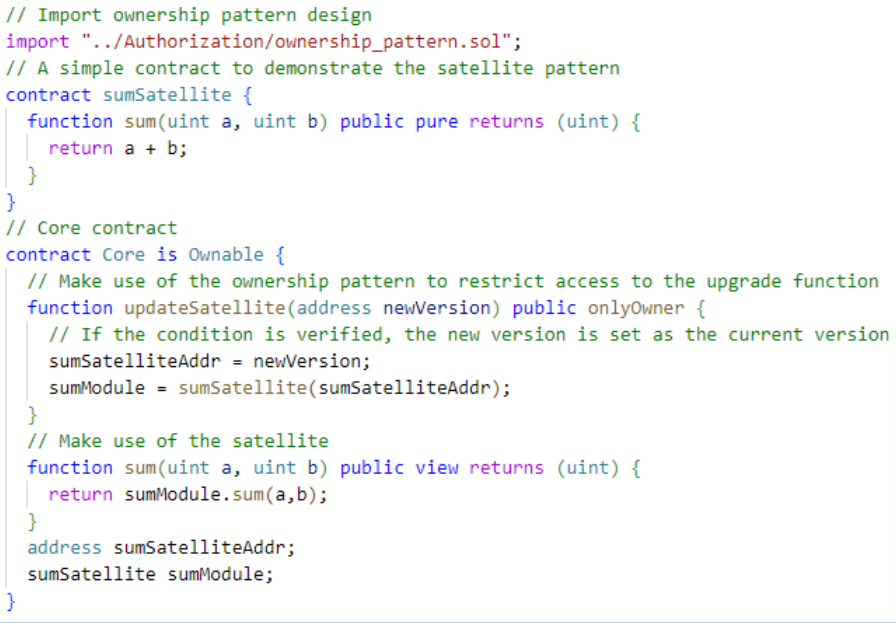
	Register Pattern
Problema	Gli utenti dello smart-contract devono interfacciarsi sempre con l'ultima versione disponibile
Soluzione	Gli utenti comunicano con lo smart-contract *Registro* per ottenere l'address dell'ultima versione dello smart-contract desiderato
Riconoscimento	Ricerca di un metodo protetto da autenticazione che aggiorni la variabile che punta all'indirizzo dello smart-contract in esecuzione e un possibile array di versioni precedenti
Snippet di Codice	
Riferimento	Qui

```
// Import ownership pattern design
import "../Authorization/ownership_pattern.sol";

contract Register is Ownable {
  // Make use of the ownership pattern to restrict access to the upgrade function
  function updateContract(address newVersion) public onlyOwner returns (bool) {
    // If both conditions are verified, the new version is set as the current version
    if(newVersion != currentVersion) {
      previousVersions.push(currentVersion);
      currentVersion = newVersion;
      return true;
    }
    return false;
  }
  // Fallback function
  fallback() external {
    (bool success, ) = currentVersion.delegatecall(msg.data);
      require(success);
  }
  address public currentVersion;
  address[] previousVersions;
}
```

SATELLITE PATTERN

Il *Satellite Pattern* ha lo scopo di definire una struttura modulare per gli smart-contract al fine di poter ovviare all'immutabilità degli stessi. Uno smart-contract è immutabile poiché per poter cambiare o aggiungere nuove funzionalità è necessario rilasciare una nuova versione.

Nello smart-contract centrale si tiene traccia di tutti i moduli disponibili e tramite un meccanismo, riservato al creatore, si può effettuare l'aggiornamento dei moduli. Poiché è necessario riconoscere il creatore dello smart-contract, l'accesso al metodo viene spesso gestito attraverso un *ownership* pattern design.

	Satellite Pattern
Problema	Gli smart-contract sono immutabili. La modifica della funzionalità richiede il rilascio di una nuova versione, ovvero un nuovo smart-contract
Soluzione	Esternalizzare e definire la logica funzionale soggetta a cambiamenti come smart-contract separati, denominati *satelliti*. Lo smart-contract *core* farà riferimento ai satelliti per usare le funzionalità necessarie alla sua esecuzione
Riconoscimento	Ricerca dei satelliti, ovvero smart-contract di piccole dimensioni che espongono funzioni con scope pubblico, e del core, lo smart-contract che contiene import dei satelliti
Snippet di Codice	
Riferimento	Qui,

```
// Import ownership pattern design
import "../Authorization/ownership_pattern.sol";
// A simple contract to demonstrate the satellite pattern
contract sumSatellite {
  function sum(uint a, uint b) public pure returns (uint) {
    return a + b;
  }
}
// Core contract
contract Core is Ownable {
  // Make use of the ownership pattern to restrict access to the upgrade function
  function updateSatellite(address newVersion) public onlyOwner {
    // If the condition is verified, the new version is set as the current version
    sumSatelliteAddr = newVersion;
    sumModule = sumSatellite(sumSatelliteAddr);
  }
  // Make use of the satellite
  function sum(uint a, uint b) public view returns (uint) {
    return sumModule.sum(a,b);
  }
  address sumSatelliteAddr;
  sumSatellite sumModule;
}
```

SECURITY DESIGN PATTERNS

I *Security design pattern* si occupano si occupa di mitigare le possibili vulnerabilità di sicurezza.

I design pattern più riconosciuti in questa tipologia sono: ***Emergency Stop (Circuit Breaker) Pattern, Rejector Pattern, Speed Bump Pattern, Mutex Pattern, Rate Limit Pattern, Balance Limit Pattern*** *e* ***Check-Effects-Interaction Pattern,***

EMERGENCY STOP (CIRCUIT BREAKER) PATTERN

L'*Emergency Stop Pattern,* spesso chiamato *Circuit Breaker Pattern*, ha lo scopo di fornire al creatore di uno smart-contract uno switch software per poter abilitare o disabilitare alcune delle sue funzionalità durante il suo runtime.

Poiché è necessario riconoscere il creatore dello smart-contract, l'accesso al metodo viene spesso gestito attraverso un *ownership* pattern design.

	Emergency Stop (Circuit Breaker) Pattern
Problema	Una volta distribuito uno smart-contract sulla rete Ethereum, esso viene eseguito autonomamente all'interno di quest'ultima. Non vi è la possibilità di interromperne l'esecuzione in caso di bug o problemi di sicurezza
Soluzione	Implementare un meccanismo a interruttore all'interno dello smart-contract che, previa autenticazione, permetta la disattivazione delle funzionalità
Riconoscimento	Ricerca di un metodo *toggle* che effettui *il flip dello stato* e relativi $modifier$ di controllo
Snippet di Codice	``` // Import ownership pattern design import "../Authorization/ownership_pattern.sol"; contract EmergencyStop is Ownable { // define the state variable bool private contractStopped = false; // Modifier to halt operations in case of emergency modifier haltInEmergency { if (!contractStopped) _; } // Modifier for emergency enabled operations modifier enableInEmergency { if (contractStopped) _; } // Toggles the emergency state function toggleContractStopped() public onlyOwner { contractStopped = !contractStopped; } // Example function to demonstrate the emergency stop pattern function deposit() public payable haltInEmergency { // some code } // Example function to demonstrate the emergency stop pattern function withdraw() public view enableInEmergency { // some code } } ```
Riferimento	Qui, Qui, Qui, Qui, Qui (Articolo IEEE)

REJECTOR PATTERN

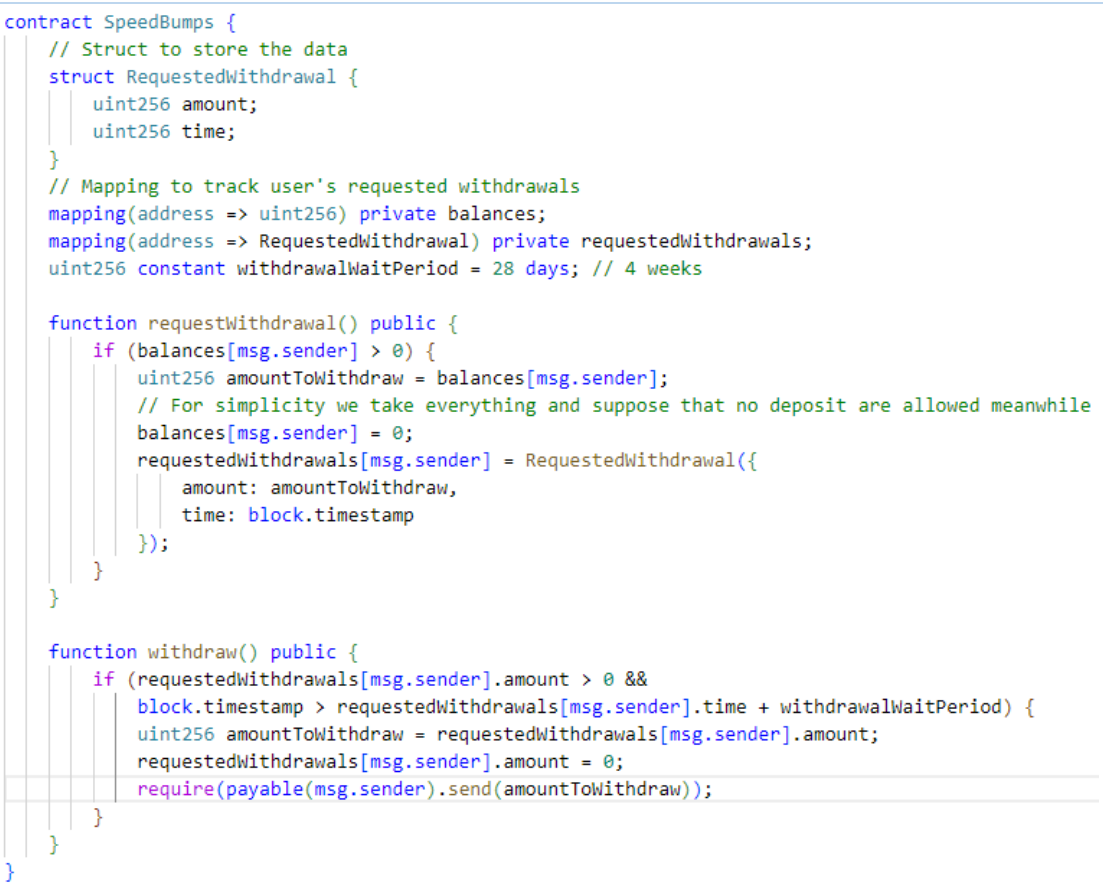
Il *Rejector Pattern* ha lo scopo di fornire al creatore di uno smart-contract un meccanismo per mantenere attivo il contratto ma rifiutare tutti i trasferimenti in entrata.

Questo design pattern viene usato in accoppiata con un design pattern che permette il disaccoppiamento fra logica e indirizzo, come ad esempio il *relay pattern*.

	Rejector Pattern
Problema	Si vuole mandare in uno stato di manutenzione temporanea lo smart-contract per poi riprendere nuovamente attività senza dover cambiare indirizzo
Soluzione	Si fa uso del *relay pattern* per sostituire la logica corrente in uno smart-contract che rifiuta ogni richiesta in entrata
Riconoscimento	Ricerca di uno smart-contract avente solo un $fallback()$ che invoca $revert()$
Note	• In versioni datate di Solidity, la funzione $fallback()$ potrebbe essere definita come $function()$; • Sempre in versioni datate, anziché $revert()$ potrebbe essere usata la keyword $throw$
Snippet di Codice	`contract Rejector {` `  fallback() external {` `  revert();` `  }` `}`
Riferimento	Qui,

SPEED BUMP PATTERN

Lo *Speed Bump Pattern* ha lo scopo di fornire un meccanismo per rallentare le esecuzione delle operazioni critiche al fine di rendere arduo il tentativo di attività fraudolente.

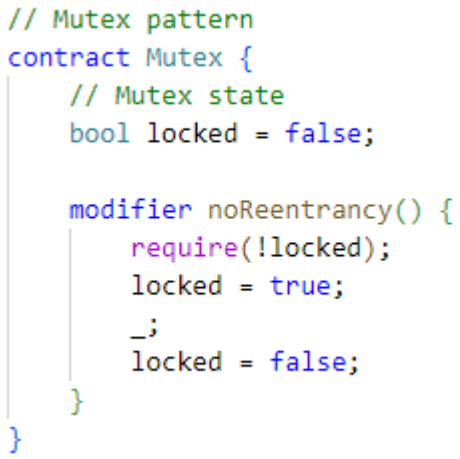
	Speed Bump Pattern
Problema	Una ripetizione massima in un lasso di tempo piccolo di operazioni critiche è in genere segno di una possibile attività fraudolenta
Soluzione	Prolungare il completamento di operazioni critiche per contrastare le attività fraudolente attraverso l'uso di intervalli d'attesa
Riconoscimento	Ricerca di una struttura dati che per utente tenga traccia del momento in cui l'utente abbia fatto richiesta dell'operazione critica e un costrutto $if\ o\ require$ nel metodo dell'operazione critica per il controllo dell'idoneità al completamento
Snippet di Codice	
Riferimento	Qui, Qui, Qui (Versione estesa), Qui (Articolo IEEE)

```
contract SpeedBumps {
    // Struct to store the data
    struct RequestedWithdrawal {
        uint256 amount;
        uint256 time;
    }
    // Mapping to track user's requested withdrawals
    mapping(address => uint256) private balances;
    mapping(address => RequestedWithdrawal) private requestedWithdrawals;
    uint256 constant withdrawalWaitPeriod = 28 days; // 4 weeks

    function requestWithdrawal() public {
        if (balances[msg.sender] > 0) {
            uint256 amountToWithdraw = balances[msg.sender];
            // For simplicity we take everything and suppose that no deposit are allowed meanwhile
            balances[msg.sender] = 0;
            requestedWithdrawals[msg.sender] = RequestedWithdrawal({
                amount: amountToWithdraw,
                time: block.timestamp
            });
        }
    }

    function withdraw() public {
        if (requestedWithdrawals[msg.sender].amount > 0 &&
            block.timestamp > requestedWithdrawals[msg.sender].time + withdrawalWaitPeriod) {
            uint256 amountToWithdraw = requestedWithdrawals[msg.sender].amount;
            requestedWithdrawals[msg.sender].amount = 0;
            require(payable(msg.sender).send(amountToWithdraw));
        }
    }
}
```

MUTEX PATTERN

Il *Mutex Pattern* ha lo scopo di mitigare le possibili vulnerabilità di rientranza, ovvero quando è una funzione viene richiamata una seconda volta, da una sequenza *funzione-chiamata esterna-funzione* oppure da sé stessa, causando un *problema inconsistenza* dei dati, in quanto la seconda esecuzione opererà prima che la prima esecuzione abbia aggiornato lo stato.

Il meccanismo proposto dal pattern si basa su un semplice blocco basato sul concetto di *semaforo* con *mutua esclusione*.

	Mutex Pattern
Problema	Si immagini una funzione che esegue un prelievo, contenente appositi controlli di validità, che richiami un'entità terza per ottenere delle informazioni. Tale entità, fraudolenta, richiama nuovamente la funzione, ottenendo un doppio prelievo al costo di uno. La seconda chiamata opera su uno stato obsoleto in quanto la prima chiamata è ancora in esecuzione. Quando il controllo dell'esecuzione ritorna alla prima chiamata, questa aggiornerà lo stato inconsapevole delle alterazioni della seconda chiamata.
Soluzione	Definire un meccanismo di blocco e sblocco di un semaforo e definire un $modifier$ che incorpori la funzionalità desiderata tramite semaforo
Riconoscimento	Ricerca del meccanismo di gestione del semaforo (semplice flip di booleano o metodi specifici) e il $modifier$
Snippet di Codice	
Riferimento	Qui (Versione complessa), Qui, Qui (Articolo IEEE)

RATE LIMIT PATTERN

Il *Rate Limit Pattern* ha lo scopo di mitigare un possibile calo di prestazioni che può affliggere uno smart-contract in seguito a una rapida successione di esecuzioni di una particolare funzionalità.

Il meccanismo proposto dal pattern si basa su un semplice blocco basato sul numero di esecuzioni effettuabili ogni quanto di tempo deciso a piacimento.

	Rate Limit Pattern
Problema	Una rapida successione di esecuzioni di una particolare funzionalità influenza le performance dello smart-contract
Soluzione	Definire un meccanismo di blocco basato sul numero di esecuzioni effettuabili ogni quanto di tempo e una funzione $reset()$
Riconoscimento	Ricerca del meccanismo di conteggio e $reset()$
Snippet di Codice	`contract RateLimiter {` `  uint256 executions;` `  uint256 enableEvery;` `  uint256 nextReset;` `  // Initialize reset time` `  constructor(uint256 _resetInterval) {` `  executions = 0;` `  enableEvery = _resetInterval;` `  nextReset = block.timestamp + _resetInterval;` `  }` `  // Reset execution count` `  function reset() private {` `  executions = 0;` `  nextReset = block.timestamp + enableEvery;` `  }` `  // Modifier to limit execution` `  modifier rateLimited(uint256 maxExecutions) {` `  if (executions++ < maxExecutions) _;` `  if (block.timestamp >= nextReset) reset();` `  }` `}`
Riferimento	Qui, Qui (Versione semplificata), Qui (Articolo IEEE)

BALANCE LIMIT PATTERN

Il *Balance Limit Pattern* ha lo scopo di contenere i possibili danni/compromissioni che possono affliggere l'utente in caso di bug del codice o l'exploit di vulnerabilità non ancora scoperte.

Il meccanismo proposto dal pattern è assai basilare: imporre un limite massimo per il bilancio delle utenze.

	Balance Limit Pattern
Problema	C'è sempre il rischio che uno smart-contract venga compromesso a causa di bug nel codice o di problemi di sicurezza ancora sconosciuti
Soluzione	Definire una quota massima di fondi, che sono a rischio, che lo smart-contract può detenere
Riconoscimento	Ricerca del $modifier$ di controllo e metodo $deposit()$
Snippet di Codice	`contract BalanceLimit {` `  uint256 limit;` `  constructor(uint256 _value) {` `  limit = _value;` `  }` `  // Modifier to check if the balance is less than the limit` `  modifier limitedPayable() {` `  require(address(this).balance <= limit);` `  _;` `  }` `  function deposit() public payable limitedPayable {` `  // Some stuff` `  }` `}`
Riferimento	Qui, Qui (Articolo IEEE)

CHECKS EFFECTS INTERACTION PATTERN

Questo pattern ha lo scopo di mitigare possibili manipolazioni di stato e dirottamenti del controllo di flusso che possono avvenire quando il nostro smart-contract comunica con un'entità, utente o smart-contract, esterno.

	Checks Effects Interaction Pattern
Problema	Quando si comunica con uno indirizzo terzo, ad esempio per trasferire Ethereum su un altro conto, il contratto chiamante trasferisce anche il flusso di controllo all'entità esterna. Questa entità esterna è ora responsabile del flusso di controllo e, nel caso si tratti di un altro contratto, può eseguire qualsiasi codice inerente. Nella maggior parte dei casi, questo meccanismo non causerà alcun problema, ma nel caso in cui l'entità abbia scopi fraudolenti potrebbe alterare il flusso di controllo e restituirlo in uno stato inaspettato al contratto iniziale. Un possibile vettore d'attacco è la rientranza.
Soluzione	Identificare le funzioni che eseguono chiamate a terzi in cui le insicurezze relative al flusso di controllo è la potenziale causa della vulnerabilità, possiamo agire di conseguenza. Come indicato nella sezione Problema, un attacco di rientranza può far sì che una funzione venga chiamata di nuovo, prima che la sua prima invocazione sia terminata. Per mitigare tale attacco non dobbiamo quindi apportare modifiche alle variabili di stato dopo aver interagito con entità esterne, poiché non possiamo fare affidamento sull'esecuzione del codice successivo all'interazione, bensì aggiornarle prima dell'interazione con l'esterno
Riconoscimento	Ricerca dei metodi di richiamo come $send()$ e $call()$ preceduti da assegnazioni di variabili
Snippet di Codice	`// Example of a bank account with balance withdrawal` `contract ChecksEffectsInteractions {` `  mapping(address => uint256) balances;` `  function deposit() public payable {` `  balances[msg.sender] = msg.value;` `  }` `  function withdraw(uint256 amount) public {` `  require(balances[msg.sender] >= amount);` `  // Update the state before sending Ether to prevent reentrancy` `  balances[msg.sender] -= amount;` `  // Perform the transfer` `  payable(msg.sender).transfer(amount);` `  }` `}`
Riferimento	Qui, Qui, Qui, Qui (Articolo IEEE)

ANALISI STATICA

Per lo sviluppo di una utility per l'analisi automatica di un codice sorgente Solidity al fine di individuare l'utilizzo dei pattern design, precedentemente descritti, si propongono le seguenti idee:

- Sviluppare una utility CLI in *Python* basata sulla libreria *python-solidity-parser*;
  - Una possibile alternativa è lo sviluppo in *NodeJS* basandosi su *solidity-parser*;
- Definire una struttura descrittiva di un Pattern Design che rappresenti la rappresentazione compatibile con l'utility sopracitata e metta a disposizione una lista di controlli per verificare l'utilizzo effettivo del design pattern;
- In caso di disponibilità di un modesto dataset, mostrare alcune statistiche usando Plotly;

ESEMPIO DI OUTPUT DEL PARSER

```
{'type': 'SourceUnit',
 'children': [{'type': 'PragmaDirective',
               'name': 'solidity',
               'value': '^0.8.0.0'},
              {'type': 'ContractDefinition'},
               'baseContracts': [],
               'kind': 'contract',
               'name': 'SimpleAuction',
               'subNodes': [{'initialValue': None,
                             'type': 'StateVariableDeclaration',
                             'variables': [{'expression': None,
                                            'isDeclaredConst': False,
                                            'isIndexed': False,
                                            'isStateVar': True,
                                            'name': 'beneficiary',
                                            'type': 'VariableDeclaration',
                                            'typeName': {'name': 'address',
                                                         'type': 'ElementaryTypeName'},
                                            'visibility': 'public'}]},
 ...
```

SOLIDITY DESIGN PATTERN ANALYZER

Solidity Design Pattern Analyzer, denominato brevemente "*analyzer",* è l'utility ideata per rispondere all'obbiettivo della tesi.

L'utility è un programma a riga di comando capace di analizzare automaticamente un file sorgente di solidity e suggerire se lo smart-contract analizzato usi o meno dei design pattern.

Il processo di analisi automatica è basato un sistema modulare fra le entità facenti parte dell'applicativo:

- Ogni design pattern che si vuole individuare deve essere descritto attraverso un *Design Pattern Descriptor*, un file *JSON* contenente una serie di controlli atti a individuare riferimenti caratteristici del design pattern sresso;
- Un *JSON-Schema* basato sullo standard *Draft-07* utilizzato per validare i descriptor forniti dall'utente

BIBLIOGRAFICA

1. Smart-Contract Patterns written in Solidity, collated for community good
2. Patterns found in smart contract oriented language Solidity
3. Smart Contracts Design Patterns in the Ethereum Ecosystem and Solidity Code
4. A compilation of patterns and best practices for the smart contract programming language Solidity
5. M. Wohrer and U. Zdun, "Smart contracts: security patterns in the ethereum ecosystem and solidity," 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), 2018, pp. 2-8, doi: 10.1109/IWBOSE.2018.8327565.
6. L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino and D. Tigano, "Design Patterns for Gas Optimization in Ethereum," 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), 2020, pp. 9-15, doi: 10.1109/IWBOSE50093.2020.9050163.