

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.

Ultimo aggiornamento: 07/11/2022

Changelog:

- Aggiunti riferimento per design pattern
- Aggiunti ulteriori Behavioral Pattern
- Aggiunti Economic Pattern
- Aggiornata bibliografica

Sommario

Classificazione Design Pattern per Smart-Contract.....	3
Authorization Design Patterns.....	3
Ownership Pattern.....	3
Access Restriction Pattern.....	4
Behavioral Design Patterns.....	5
Pull Payment (<i>Pull over Push</i>) Pattern.....	5
State Machine Pattern.....	6
Oracle Pattern.....	7
Commit & Reveal Pattern.....	8
Randomness Pattern.....	9
Guard Check Pattern.....	10
Lifecycle Design Patterns.....	11
Mortal Pattern.....	11
Auto Deprecation Pattern.....	12
Maintenance Design Patterns.....	12
Data Segregation (Eternal Storage) Pattern.....	13
Relay (Proxy Delegate) Pattern.....	13
Register Pattern.....	14
Satellite Pattern.....	15
Security Design Patterns.....	16
Emergency Stop (Circuit Breaker) Pattern.....	16
Rejector Pattern.....	16
Speed Bump Pattern.....	17
Mutex Pattern.....	18
Rate Limit Pattern.....	19
Balance Limit Pattern.....	19
Checks Effects Interaction Pattern.....	20
Bibliografica.....	22

Classificazione Design Pattern per Smart-Contract

I design pattern impiegati nello sviluppo di smart-contract possono essere classificati in diverse tipologie, ognuna aventi uno scopo ben preciso:

- *Authorization*: questa tipologia di design pattern si occupa di gestire l'accesso alle funzionalità di uno smart-contract;
- *Behavioral*: questa tipologia di design pattern si occupa di fornire funzionalità e meccanismi per lo svolgimento delle operazioni di uno 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 agli 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

Questo 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 <i>modifier</i> con costruito <i>if</i> o <i>require</i> che confronti <i>msg.sender</i> con l'indirizzo del proprietario memorizzato

<p>Snippet di Codice</p>	<pre> 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; } } </pre>
<p>Riferimenti</p>	<p>Qui, Qui, Qui (Versione riportata)</p>

Access Restriction Pattern

Questo pattern ha lo scopo di aggiungere dei controlli di prerequisiti, che nel comportamento predefinito di Solidity non sono contemplati, per l'esecuzione dei metodi dello smart-contract.

Access Restriction Pattern	
<p>Problema</p>	<p>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</p>
<p>Soluzione</p>	<p>Definire una serie di <i>modifier</i> generalmente applicabili che controllano i prerequisiti desiderati e applicarli nella definizione dei metodi</p>
<p>Riconoscimento</p>	<p>Ricerca di una serie <i>modifier</i> all'interno dello smart-contract</p>

Snippet di Codice	<pre> 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 } } </pre>
Riferimento	Qui (Versione estesa) , Qui , Qui

Behavioral Design Patterns

Gli *Behavioral design pattern* si occupano di fornire funzionalità e meccanismi 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

Questo pattern ha lo scopo di fornire un meccanismo per eseguire pagamenti (trasferimenti di criptovaluta) in modo sicuro.

Full Payment (<i>Pull over Push</i>) 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 <i>fallback()</i> 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 <i>sender</i> : <i>msg.sender.transfer()</i>

Snippet di Codice	<pre>// 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); } }</pre>
Riferimento	Qui , Qui

State Machine Pattern

Questo pattern ha lo scopo di fornire un meccanismo per consentire a uno smart-contract di passare 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 <i>enum</i> che contenga gli stati e il <i>modifier</i> che controlla lo stato

<p>Snippet di Codice</p>	<pre>// 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); } }</pre>
<p>Riferimento</p>	<p>Qui</p>

Oracle Pattern

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

Oracle Pattern	
<p>Problema</p>	<p>Si immagini un caso d'uso in cui lo smart-contract necessita 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</p>
<p>Soluzione</p>	<p>Richiedere i dati esterni attraverso un servizio <i>oracolo</i> che è collegato al mondo esterno e funge da provider di dati</p>
<p>Riconoscimento</p>	<p>Ricerca di un <i>oracolo</i> ovvero di uno smart-contract contenente una <i>struct</i> per le richieste e i metodi <i>query</i> e <i>reply</i></p>

Snippet di Codice	<pre> // 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 } } </pre>
Riferimento	Qui , Qui (Versione basata su Oraclize)

Commit & Reveal Pattern

Questo pattern ha lo scopo di fornire un meccanismo per consentire a uno smart-contract di passare attraverso diverse fasi con diverse funzionalità corrispondenti esposte.

Commit & Reveal Pattern	
Problema	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
Soluzione	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 <i>commit</i> in cui viene scelto e specificato un valore e una fase di <i>reveal</i> in cui il valore viene rivelato e controllato
Riconoscimento	Ricerca dei metodi <i>commit</i> e <i>reveal</i> , dei log <i>LogCommit</i> e <i>LogReveal</i> e relativi <i>emit</i> , <i>struct</i> dei commit

<p>Snippet di codice</p>	<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"; // committed 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>
<p>Riferimento</p>	<p>Qui</p>

Randomness Pattern

Questo 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	
<p>Problema</p>	<p>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.</p> <p>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.</p> <p>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.</p>
<p>Soluzione</p>	<p>Sono state ideate diverse soluzioni per superare queste limitazioni, ciascuna con i rispettivi vantaggi e svantaggi:</p> <ul style="list-style-type: none"> • <i>Block hash PRNG</i>: l'hash del blocco viene usato come sorgente di casualità; • <i>Oracle RNG</i>: si usa un oracolo il quale fa provider per la casualità;

	<ul style="list-style-type: none"> • <i>Collaborative PRNG</i>: una generazione collaborativa di casualità dentro la blockchain;
Riconoscimento	<ul style="list-style-type: none"> • Per individuare l'implementazione più semplice di randomness basta individuare: <ul style="list-style-type: none"> ➤ <code>return uint(blockhash(block.number - 1))</code>, da non utilizzare per scommesse in quanto è facilmente determinabile; ➤ <code>return uint(keccak256(abi.encodePacked(block.timestamp, msg.sender, block.difficulty)))</code> • Le altre implementazioni sono ostiche da analizzare
Snippet di codice	<pre>contract SimpleRandom { // Simple random number generator function simpleRandomNumber() internal view returns (uint) { return uint(blockhash(block.number - 1)); } // 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))); } }</pre>
Riferimento	Qui (Spiegazione e versione estesa)

Guard Check Pattern

Questo 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	<p>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.</p> <p>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.</p>
Soluzione	<p>Per raggiungere questi obiettivi, 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.</p>
Riconoscimento	<p>Per riconoscere tale pattern si ricerca l'utilizzo di:</p> <ul style="list-style-type: none"> • <code>assert(condition)</code>: usato solo per verificare la presenza di errori interni e per controllare gli invarianti (asserzioni sempre <i>true</i>); <ul style="list-style-type: none"> ➤ Rimborsa tutto il gas che non è stato consumato nel momento in cui viene lanciata l'eccezione; • <code>require(condition)</code>: 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; <ul style="list-style-type: none"> ➤ Consuma tutto il gas incluso nella transazione • <code>revert()</code>: usato per ripristinare le modifiche apportate allo stato, viene normalmente lanciato in caso di fallimento da <i>require</i> o <i>assert</i> ma potrebbe essere utilizzato all'interno di un controllo fatto da un costrutto <i>if</i>.

Snippet di codice	<pre>// Guard check pattern contract GuardCheck { function donate(address addr) payable public { require(addr != address(0)); require(msg.value != 0); uint balanceBeforeTransfer = address(this).balance; uint transferAmount; if (addr.balance == 0) { transferAmount = msg.value; } else if (addr.balance < msg.sender.balance) { transferAmount = msg.value / 2; } else { // revert if the condition is not met revert(); } payable(addr).transfer(transferAmount); // Check that the transfer was successful assert(address(this).balance == balanceBeforeTransfer - transferAmount); } }</pre>
Riferimento	Qui (Spiegazione)

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 <i>selfdestruct</i> ([address]) all'interno di un metodo che esegue un controllo preliminare dell'autorizzazione del richiedente
Note	<ul style="list-style-type: none"> Per smart-contract di vecchia data anziché <i>selfdestruct</i>([address]) è possibile trovare <i>suicide</i>([address]). ➤ Riferimento: EIP6 Sempre per vecchie versioni, la funzione canonica di terminazione prende il nome di <i>kill</i>() anziché <i>destroy</i>()
Riconoscimento	Ricerca di un costrutto <i>if</i> o <i>require</i> o un <i>modifier</i> per l'autenticazione seguito dalla chiamata <i>selfdestruct</i> ([address])

Snippet di Codice	<pre>// 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())); } } }</pre>
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 interrompi 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	<ul style="list-style-type: none"> Da Solidity 0.7.0 la <i>keyword now</i> è stata deprecata in favore di <i>block.timestamp</i>;
Riconoscimento	Ricerca della funzione <i>expired()</i> e <i>modifier</i> che fanno uso di tale funzione
Snippet di Codice	<pre>contract Deprecatable { // Initialize the contract with a deprecation date constructor(uint _lifetime) { expires = block.timestamp + _lifetime; } // Check if the contract is still valid function expired() public view returns (bool) { return block.timestamp > expires ? true : false; } // Modifier to perform when the contract is still valid modifier willDeprecate() { if (!expired()) {_;} } // Modifier to perform when the contract is no longer valid modifier whenDeprecated() { if (expired()) {_;} } uint expires; }</pre>
Riferimento	Qui , Qui , Qui

Maintenance Design Patterns

I *maintenance 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 (EternalStorage) Pattern*, *Relay (Proxy Delegate) Pattern*, *Satellite Pattern* e *Register Pattern*.

Data Segregation (Eternal Storage) Pattern

Questo 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 <i>Storage</i> che esponga eventuali <i>getter</i> e <i>setter</i> . Possibile implementazione del <i>Proxy</i> .
Snippet di Codice	<pre>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) {} }</pre>
Riferimento	Qui , Qui , Qui , Qui (Versione estesa dello storage)

Relay (Proxy Delegate) Pattern

Questo 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 <i>proxy</i> che inoltrerà tutte le richieste alla più recente versione dello smart-contract desiderato
Note	<ul style="list-style-type: none"> In versioni datate di Solidity, la funzione <i>fallback()</i> potrebbe essere definita come <i>function()</i>;
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 <i>fallback()</i>
Snippet di Codice	<pre>// 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; }</pre>
Riferimento	Qui , Qui , Qui (Versione estesa) , Qui (Versione ottimizzata per gas)

Register Pattern

Questo 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 pressoché 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 <i>Registro</i> 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	<pre> // 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; } </pre>
Riferimento	Qui

Satellite Pattern

Questo 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 viene detto immutabile poiché per poter cambiare o aggiungere nuove funzionalità allo smart-contract 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 <i>satelliti</i> . Lo smart-contract <i>core</i> 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	<pre> // 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; } </pre>

Riferimento	Qui ,
-------------	-----------------------

Security Design Patterns

I *Security design pattern* si occupano 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

Questo 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 all'interno dello smart-contract che previa autenticazione permetta la disattivazione delle funzionalità
Riconoscimento	Ricerca di un metodo <i>toggle</i> che effettui il <i>flip dello stato</i> e relativi <i>modifier</i> di controllo
Snippet di Codice	<pre>// 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 } }</pre>
Riferimento	Qui , Qui , Qui , Qui ,

Rejector Pattern

Questo 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 <i>relay pattern</i> per sostituire la logica corrente in uno smart-contract che rifiuta ogni richiesta in entrata
Riconoscimento	Ricerca di uno smart-contract avente solo un <i>fallback()</i> che invoca <i>revert()</i>
Note	<ul style="list-style-type: none"> • In versioni datate di Solidity, la funzione <i>fallback()</i> potrebbe essere definita come <i>function()</i>; • Sempre in versioni datate, anziché <i>revert()</i> potrebbe essere usata la keyword <i>throw</i>
Snippet di Codice	<pre>contract Rejector { fallback() external { revert(); } }</pre>
Riferimento	Qui ,

Speed Bump Pattern

Questo pattern ha lo scopo di fornire un meccanismo rallentamento dell'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 <i>if o require</i> nel metodo dell'operazione critica per il controllo dell'idoneità al completamento

<p>Snippet di Codice</p>	<pre> 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)); } } } </pre>
<p>Riferimento</p>	<p>Qui, Qui, Qui (Versione estesa)</p>

Mutex Pattern

Questo pattern ha lo scopo di mitigare le possibili vulnerabilità di rientranza, ovvero quando è possibili richiamare per una seconda volta un metodo prima ancora 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	
<p>Problema</p>	<p>Si immagini una funzione che esegue un prelievo, contenente appositi controlli di validità, che venga richiamata dall'utente 2 volte consecutive. La seconda esecuzione potrebbe operare su uno stato obsoleto e non aggiornato poiché la prima esecuzione non è ancora finita</p>
<p>Soluzione</p>	<p>Definire un meccanismo di blocco e sblocco di un semaforo e definire un <i>modifier</i> che incorpori la funzionalità desiderata tramite semaforo</p>
<p>Riconoscimento</p>	<p>Ricerca del meccanismo di gestione del semaforo (semplice flip di booleano o metodi specifici) e il <i>modifier</i></p>

Snippet di Codice	<pre>// Mutex pattern contract Mutex { // Mutex state bool locked = false; modifier noReentrancy() { require(!locked); locked = true; _; locked = false; } }</pre>
Riferimento	Qui (Versione completa) , Qui

Rate Limit Pattern

Questo 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 affliggere le performance dello smart-contract
Soluzione	Definire un meccanismo di blocco basato sul numero di esecuzioni effettuabili ogni quanto di tempo e una funzione <code>reset()</code>
Riconoscimento	Ricerca del meccanismo di conteggio e <code>reset()</code>
Snippet di Codice	<pre>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(); } }</pre>
Riferimento	Qui , Qui (Versione semplificata) ,

Balance Limit Pattern

Questo 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 <i>modifier</i> di controllo e metodo <i>deposit()</i>
Snippet di Codice	<pre>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 } }</pre>
Riferimento	Qui ,

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 un 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	Identificate le funzioni che eseguono chiamate a terzi in cui le sue 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 <i>send()</i> e <i>call()</i> preceduti da assegnazioni di variabili

Snippet di Codice	<pre>// 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); } }</pre>
Riferimento	Qui , Qui , Qui

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.](#)