TRUST: un'Applicazione di Divisione Spese Decentralizzata su Ethereum

Report Finale di Progetto

Federico Paccosi, Alessio Milano

Luglio 2025

Sommario

Questo report descrive la progettazione e l'implementazione di "TRUST", un'applicazione decentralizzata (DApp) basata sulla blockchain di Ethereum, che replica ed estende le funzionalità del popolare servizio di divisione delle spese Splitwise. Il documento analizza in modo approfondito le decisioni architetturali chiave, valuta i costi computazionali (gas), esamina in dettaglio le misure di sicurezza adottate e fornisce un manuale per il deploy e l'interazione con gli smart contract sviluppati. Il progetto è stato realizzato utilizzando il framework Hardhat e il linguaggio di programmazione Solidity, con un'enfasi particolare sull'efficienza del gas e sulla robustezza del codice.

Indice

1 Introduzione

Il presente documento illustra la progettazione, l'implementazione e l'analisi di TRUST, un'applicazione decentralizzata (DApp) per la gestione e la divisione delle spese, concepita come un'alternativa basata su blockchain al servizio Splitwise. Sviluppato sulla blockchain di Ethereum, il sistema sfrutta le proprietà intrinseche dei distributed ledger per offrire un ambiente caratterizzato da trasparenza, immutabilità e *trustlessness*.

L'implementazione su blockchain offre vantaggi significativi rispetto alle soluzioni centralizzate tradizionali:

- **Trasparenza:** Tutte le transazioni e le registrazioni delle spese sono pubbliche e verificabili da ogni membro del gruppo, eliminando ogni ambiguità.
- **Immutabilità:** Una volta che una spesa è registrata sulla blockchain, non può essere alterata o cancellata, fornendo una cronologia a prova di manomissione utile per la risoluzione di dispute.
- **Pagamenti Nativi:** Gli utenti possono saldare i propri debiti direttamente on-chain utilizzando un token, senza la necessità di intermediari finanziari esterni come banche o PayPal, facilitando le transazioni transfrontaliere.

L'obiettivo primario del progetto è stato quello di creare un ecosistema peer-to-peer dove gli utenti potessero formare gruppi, registrare spese condivise, semplificare i flussi di debito e saldare i conti direttamente on-chain, attraverso l'uso di un token fungibile conforme allo standard ERC-20. L'interazione con i contratti è gestita tramite script Hardhat, che permettono di automatizzare il deploy e di simulare scenari di utilizzo complessi.

Questo report si propone di documentare in modo esaustivo l'intero ciclo di vita dello sviluppo: dalle scelte architetturali che hanno guidato la stesura degli smart contract, all'analisi dettagliata del codice sorgente, fino alla valutazione delle performance in termini di costo del gas, all'analisi delle vulnerabilità e alla stesura di un manuale operativo. Lo stack tecnologico impiegato comprende Solidity per la logica on-chain, il framework Hardhat per lo sviluppo e il testing, e le librerie OpenZeppelin per garantire sicurezza e aderenza agli standard.

2 Setup del Progetto e Ambiente di Sviluppo

Per lo sviluppo del progetto è stato utilizzato il framework **Hardhat**, come richiesto dalle specifiche. Hardhat è stato scelto per il suo ambiente di sviluppo completo, che integra compilazione, testing, debugging e deployment in un unico workflow efficiente. L'utilizzo di **TypeScript** ha inoltre fornito una maggiore robustezza al codice degli script e dei test grazie alla tipizzazione statica, che previene errori comuni in fase di sviluppo.

La struttura del progetto, generata da Hardhat, è la seguente:

- `/contracts`: Contiene i sorgenti Solidity dei nostri smart contract (`TrustManager.sol`, `TrustToken.sol`).
- `/scripts`: Contiene gli script di interazione, come `interaction.ts`, per eseguire scenari di utilizzo sulla blockchain.
- `/test`: Contiene i file di test unitari e di integrazione per i contratti (`TrustManager.test.ts`, `TrustToken.test.ts`).
- `hardhat.config.ts`: File di configurazione di Hardhat, dove sono state definite le versioni del compilatore e le impostazioni per i plugin, come `hardhat-gas-reporter`.

L'ambiente di sviluppo è stato inizializzato con i comandi standard di Node.js e Hardhat:

```
mkdir trust-project && cd trust-project
npm init -y
npm install --save-dev hardhat
npx hardhat init # Scegliendo un progetto TypeScript
npm install --save-dev @nomicfoundation/hardhat-toolbox hardhat-gas-reporter
```

3 Analisi del Codice e delle Scelte Implementative

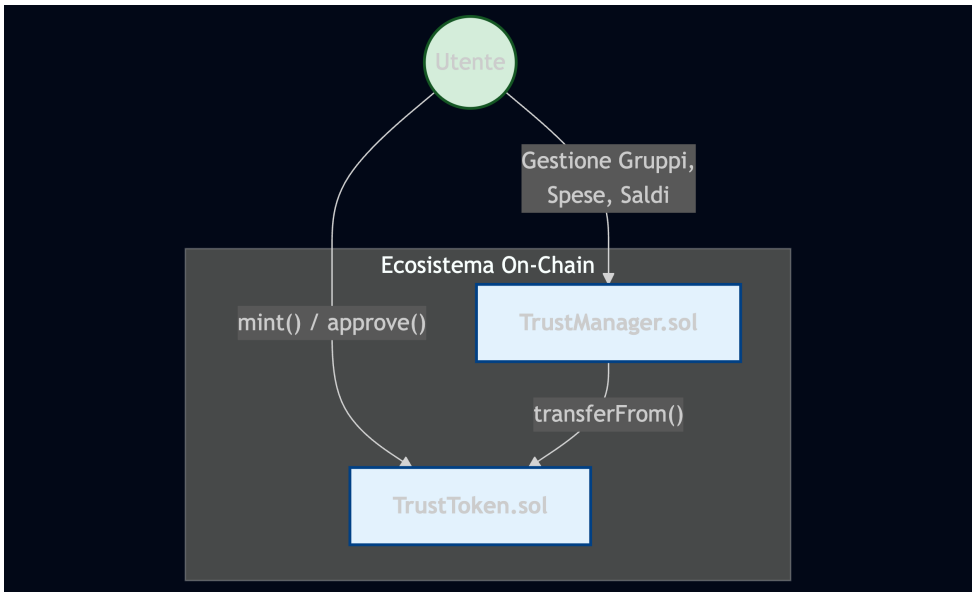

Figura 1: Diagramma dell'architettura di alto livello del sistema TRUST. Mostra le interazioni logiche tra l'utente e i due smart contract.

3.1 Analisi del Contratto `TrustManager.sol`

Il contratto `TrustManager.sol` rappresenta il nucleo logico dell'applicazione. È responsabile della gestione dei gruppi, della contabilità delle spese e dei saldi, e dell'orchestrazione delle operazioni di semplificazione e saldo dei debiti.

3.1.1 Strutture Dati e Variabili di Stato

Una progettazione efficiente dello storage è fondamentale per minimizzare i costi di gas.

- **Indirizzo del Token:** L'indirizzo del contratto `TrustToken` è memorizzato in una variabile `immutable`.

```
address public immutable trustTokenAddress;
```

  A differenza di una variabile di stato standard, il cui valore viene letto dallo storage ad ogni accesso (operazione `SLOAD`, costosa), il valore di una variabile `immutable` viene impostato una sola volta nel costruttore e "bruciato" direttamente nel bytecode del contratto. Questo rende ogni lettura successiva efficiente quasi quanto una costante, ottimizzando significativamente il gas.

- **Structs:** Sono state definite due `struct` principali per aggregare logicamente i dati: `Group` e `Expense`. Una terza struct di supporto, `BalanceInfo`, è usata solo in memoria per l'algoritmo di semplificazione.

```
struct Group {
    address owner;
    address[] members;
}

struct Expense {
    uint256 id;
    string description;
    uint256 totalAmount;
    address payer;
    uint256 groupId;
    uint256 timestamp;
}

```

Listing 1: Definizione delle struct

- **Gestione degli ID:** Contatori `private` come `nextGroupId` e `nextExpenseId` sono utilizzati per generare ID univoci e sequenziali, un pattern semplice e robusto per l'identificazione delle risorse.
- **Eventi:** Per ogni azione che modifica lo stato in modo significativo, viene emesso un evento (`GroupCreated`, `ExpenseAdded`, etc.). L'uso di `indexed` per i parametri chiave (es. `groupId`, `user`) è una scelta deliberata per permettere alle applicazioni off-chain di filtrare e cercare questi eventi in modo efficiente, senza dover processare l'intera cronologia dei log.

3.1.2 La Scelta Architetturale Critica: Gestione dei Saldi

Questa è stata la decisione di design più importante dell'intero progetto, un bivio che ha determinato la fattibilità e l'efficienza economica dell'intera applicazione on-chain.

L'Approccio Ingenuo e la "Bomba di Gas" L'algoritmo di semplificazione dei debiti, come da specifica, richiede come primo passo il calcolo del saldo netto di ogni membro del gruppo. L'approccio più intuitivo e diretto sarebbe stato quello di modellare i debiti come un grafo, usando un mapping annidato:

```
// APPROCCIO SCARTATO E INEFFICIENTE
mapping(address => mapping(address => uint256)) public debts;
```

Il problema di questa struttura è che, per calcolare il saldo netto di un utente 'U', il contratto avrebbe dovuto ciclare su tutti gli altri 'N' membri del gruppo per sommare sia i suoi crediti ('debts[M][U]') che i suoi debiti ('debts[U][M]'). Questa operazione, con una complessità computazionale di $O(N^2)$ per l'intero gruppo, si sarebbe tradotta in un numero proibitivo di letture dallo storage. Poiché ogni lettura ('SLOAD') sulla EVM ha un costo in gas significativo (come verrà approfondito nella Sezione 6), la funzione di semplificazione sarebbe diventata una vera e propria "bomba di gas", rendendola economicamente insostenibile e inutilizzabile su larga scala.

La Soluzione: Saldi Netti e il Trade-Off Vincente Per evitare la "bomba di gas", abbiamo scartato l'approccio del grafo e abbiamo adottato una strategia basata su un **trade-off ingegneristico**: spendere un po' più di gas subito per risparmiare un'enorme quantità di gas dopo. Invece di calcolare i saldi al bisogno, li memorizziamo e li aggiorniamo incrementalmente ogni volta che viene aggiunta una spesa. Abbiamo implementato un singolo mapping che tiene traccia del saldo netto di ogni utente:

```
// Struttura dati adottata, ultra-efficiente in lettura
mapping(uint256 => mapping(address => int256)) public balances;

// Logica di aggiornamento all'interno della funzione _addExpense(...)
function _addExpense(...) internal {
    // ...
    // Il pagatore e' creditore per l'intero importo
    balances[_groupId][_payer] += int256(_totalAmount);

    // Ogni partecipante diventa debitore per la sua quota
    for (uint i = 0; i < _debtors.length; i++) {
        balances[_groupId][_debtors[i]] -= int256(_amounts[i]);
    }
    // ...
}
```

Listing 2: Mapping dei saldi netti e logica di aggiornamento

Il vantaggio di questa architettura è devastante: quando la funzione 'simplifyDebts' viene chiamata, non deve eseguire alcun calcolo complesso. Il suo primo passo si riduce a una semplice e ultra-efficiente lettura (*O(N)*) dei saldi già pronti dal mapping. L'uso di 'int256' è essenziale per poter rappresentare sia i crediti (valori positivi) che i debiti (valori negativi).

La Prova: Stima Teorica del Risparmio Per dare una dimensione concreta al problema, possiamo stimare il costo in gas dell'approccio scartato. Come abbiamo visto, il calcolo del saldo netto di ogni utente richiederebbe di scorrere l'intero gruppo per sommare crediti e debiti.

Per capire da dove viene la stima dei costi, analizziamo un esempio pratico. In un gruppo di 3 persone (N=3, Alice, Bob, Charlie), per calcolare il saldo di tutti, il contratto dovrebbe eseguire le seguenti letture:

- **Per Alice:** Leggere `debts[Bob][Alice]`, `debts[Charlie][Alice]` (crediti) e `debts[Alice][Bob]`, `debts[Alice][Charlie]` (debiti). Totale: 4 letture.
- **Per Bob:** Leggere `debts[Alice][Bob]`, `debts[Charlie][Bob]` e `debts[Bob][Alice]`, `debts[Bob][Charlie]`. Totale: 4 letture.
- **Per Charlie:** Leggere `debts[Alice][Charlie]`, `debts[Bob][Charlie]` e `debts[Charlie][Alice]`, `debts[Charlie][Bob]`. Totale: 4 letture.

Il totale è di 12 letture, che corrisponde esattamente alla formula '2 * N * (N-1)' (ovvero '2 * 3 * 2 = 12'). Per un gruppo di 10 membri, questo calcolo si traduce in ben **180 letture** dallo storage.

È fondamentale sottolineare che questo costo enorme non verrebbe sostenuto solo la prima volta, ma **ogni singola volta** che la funzione 'simplifyDebts' venisse chiamata in una nuova transazione. Il concetto di storage "a freddo" (più costoso) e "a caldo" (più economico) viene infatti azzerato per ogni transazione. La EVM non ha memoria delle esecuzioni passate; richiamare la funzione oggi farebbe ripartire il processo di lettura da zero, con tutti i costi associati. Poiché ogni coppia '(UtenteA, UtenteB)' corrisponde a uno slot di memoria unico, quasi tutte le 180 letture sarebbero "a freddo".

Considerando un costo di 2100 gas per ogni lettura "a freddo" (`SLOAD`), il solo costo delle letture sarebbe:

$$180 \text{ letture "a freddo"} \times 2100 \text{ gas/lettura} = 378.000 \text{ gas}$$

A questo andrebbe aggiunto il costo dei loop e delle somme, portando facilmente il costo totale della funzione a **oltre 400.000 gas**. Confrontando questa stima con il costo effettivo che è stato misurato per la nostra funzione ottimizzata (un dato analizzato nel dettaglio nella Sezione 6), è evidente che il refactoring ha ridotto il consumo di gas in modo drastico, trasformando un'operazione insostenibile in una delle più efficienti del contratto.

La Controprova: l'Efficienza del Refactoring in Pratica Per completare l'analisi, esaminiamo come controprova il comportamento del sistema attuale, quello con il refactoring. Grazie alla scelta di mantenere i saldi netti sempre aggiornati, il primo passo della funzione `simplifyDebts` diventa incredibilmente leggero.

Riprendiamo lo stesso esempio pratico di un gruppo di 3 persone (N=3, Alice, Bob, Charlie). Quando `simplifyDebts` viene chiamata, il contratto non deve eseguire alcun calcolo complesso, ma solo leggere i saldi già pronti:

- Legge il saldo di Alice da `balances[groupId][Alice]` (1 lettura `SLOAD`).
- Legge il saldo di Bob da `balances[groupId][Bob]` (1 lettura `SLOAD`).
- Legge il saldo di Charlie da `balances[groupId][Charlie]` (1 lettura `SLOAD`).

Il totale è di sole **3 letture** (*O(N)*), in netto contrasto con le 12 dell'approccio precedente. Il costo in gas per questa fase iniziale si riduce a:

$$3 \text{ letture "a freddo"} \times 2100 \text{ gas/lettura} = 6.300 \text{ gas}$$

Questo costo di partenza estremamente basso è il motivo per cui l'intera esecuzione della funzione `simplifyDebts` si attesta su una media di soli 45.006 gas. Questa controprova dimostra in modo inequivocabile come la scelta architetturale di aggiornare i saldi incrementalmente sia stata la chiave per trasformare una funzione teoricamente costosissima in un'operazione pratica, veloce ed economicamente vantaggiosa per l'utente.

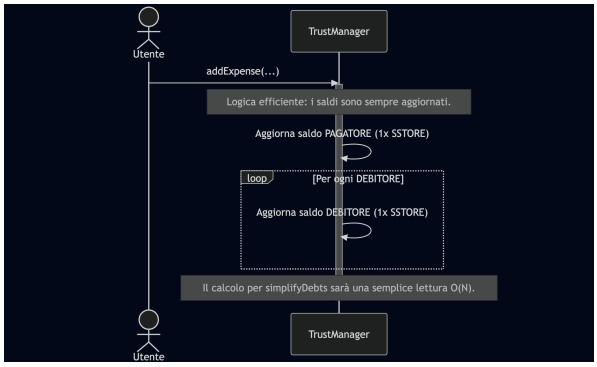

Figura 2: Flusso con l'approccio efficiente: i saldi vengono aggiornati incrementalmente durante l'aggiunta di una spesa.

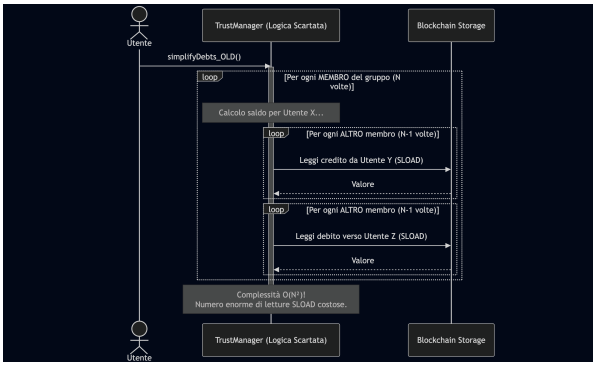

Figura 3: Flusso con l'approccio inefficiente: la semplificazione richiederebbe un calcolo complesso e un numero elevato di letture dallo storage.

3.1.3 Analisi delle Funzionalità Principali

Gestione dei Gruppi (`createGroup`, `joinGroup`) Queste funzioni gestiscono il ciclo di vita dei gruppi. La funzione `createGroup` accetta un array di membri iniziali come `calldata` per ottimizzare il gas e previene l'aggiunta di membri duplicati tramite la funzione helper `_isMemberOf`. L'uso di un puntatore `storage` all'array dei membri in `_isMemberOf` è un'ottimizzazione che evita di copiare l'intero array in memoria durante l'iterazione.

```
function createGroup(address[] calldata _initialMembers) external {
    uint256 groupId = nextGroupId;
    Group storage newGroup = groups[groupId];
    newGroup.owner = msg.sender;
    newGroup.members.push(msg.sender);
    userGroups[msg.sender].push(groupId);
    for (uint i = 0; i < _initialMembers.length; i++) {
```

```
        address member = _initialMembers[i];
        if (member != address(0) && !_isMemberOf(groupId, member)) {
            newGroup.members.push(member);
            userGroups[member].push(groupId);
        }
    }
    emit GroupCreated(groupId, msg.sender, newGroup.members);
    nextGroupId++;
}
```

Listing 3: Funzione createGroup

Gestione delle Spese (`addExpense...`) Per aderire alla specifica, sono state implementate tre funzioni separate (`addExpenseEqually`, `addExpenseByPercentage`, `addExpenseWithExactAmounts`). Per evitare la duplicazione del codice, la logica comune di validazione e registrazione è stata incapsulata nella funzione `internal _addExpense`. Nelle funzioni per divisione equa e percentuale, viene gestito esplicitamente il resto della divisione intera per garantire l'integrità contabile e non perdere fondi.

```
function addExpenseEqually(
    uint256 _groupId,
    string calldata _description,
    uint256 _totalAmount,
    address[] calldata _participants
) external {
    require(_participants.length > 0, "Must have at least one participant");

    uint256 share = _totalAmount / _participants.length;
    uint256 remainder = _totalAmount % _participants.length;

    // ... preparazione degli array ...

    // Per evitare perdite di fondi, il resto viene assegnato al primo
    partecipante.
    if (remainder > 0) {
        amounts[0] += remainder;
    }

    _addExpense(_groupId, _description, _totalAmount, msg.sender, debtors,
    amounts);
}
```

Listing 4: Esempio: addExpenseEqually e gestione del resto

Semplificazione dei Debiti (`simplifyDebts`) Questa funzione implementa l'algoritmo greedy. I passaggi chiave sono:

1. **Partizionamento:** Lettura dei saldi netti (efficiente grazie al refactoring) e suddivisione in due array in memoria, `debtors` e `creditors`.
2. **Ordinamento:** Invocazione di `_sortBalances`, che implementa *Insertion Sort*. La scelta di questo algoritmo è un compromesso tra semplicità di implementazione in Solidity e performance, che sono adeguate per gruppi di dimensioni ridotte. Per DApp su larga scala, questo diventerebbe un collo di bottiglia e si dovrebbero esplorare alternative.
3. **Reset e Ricostruzione:** I saldi del gruppo vengono azzerati e poi ricostruiti secondo il nuovo grafo semplificato determinato dal "greedy matching".

La funzione helper `_trimBalanceArray` utilizza un blocco `assembly` per manipolare direttamente la lunghezza di un array in memoria, un'operazione a basso livello molto efficiente in termini di gas.

Saldo dei Debiti (`settleDebt`) Questa funzione orchestra il pagamento on-chain e rappresenta un punto critico per la sicurezza.

- **Pattern Approve/TransferFrom:** L'interazione si basa sul meccanismo standard di ERC-20, dove un utente prima approva una spesa e poi il contratto la esegue. Questo permette al `TrustManager` di agire come intermediario fidato per aggiornare lo stato interno contestualmente al pagamento.
- **Calcolo dell'Importo:** L'importo da saldare è il minimo tra il debito e il credito, permettendo pagamenti parziali in scenari complessi con più creditori/debitori.
- **Sicurezza:** La funzione è un esempio pratico del pattern **Checks-Effects-Interactions**, come verrà discusso nella sezione delle vulnerabilità.

```
function settleDebt(uint256 _groupId, address _creditorAddress) external {
    address debtor = msg.sender;

    int256 debtorBalance = balances[_groupId][debtor];
    int256 creditorBalance = balances[_groupId][_creditorAddress];
    require(debtorBalance < 0, "SettleDebt: You do not have a negative balance.");
    require(creditorBalance > 0, "SettleDebt: The specified user is not a creditor.");

    uint256 amountToSettle = min(abs(debtorBalance), uint256(creditorBalance));
    require(amountToSettle > 0, "SettleDebt: No debt to settle between these users.");

    balances[_groupId][debtor] += int256(amountToSettle);
    balances[_groupId][_creditorAddress] -= int256(amountToSettle);

    IERC20 token = IERC20(trustTokenAddress);
    bool success = token.transferFrom(debtor, _creditorAddress, amountToSettle);
    require(success, "SettleDebt: ERC20 transfer failed. Check allowance.");

    emit DebtSettled(_groupId, debtor, _creditorAddress, amountToSettle);
}
```

Listing 5: Funzione settleDebt

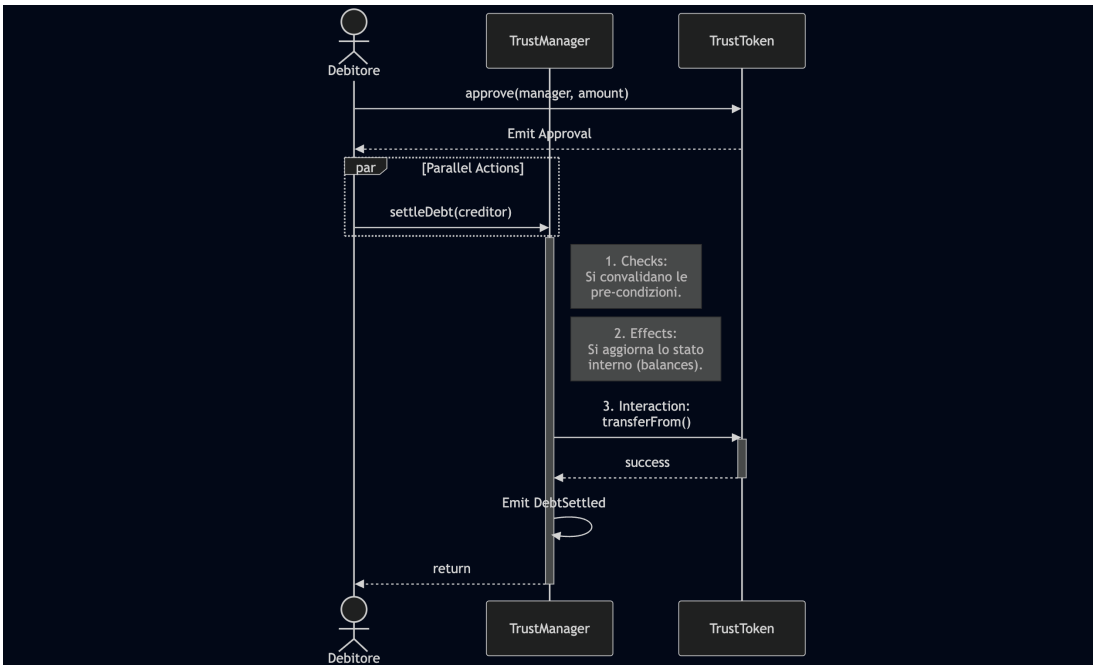

Figura 4: Diagramma di sequenza UML per la funzione `settleDebt`. Viene evidenziata l'applicazione rigorosa del pattern di sicurezza **Checks-Effects-Interactions** attraverso note esplicative.

3.2 Analisi del Contratto `TrustToken.sol`

Il contratto `TrustToken.sol` implementa un token fungibile conforme allo standard ERC-20, che funge da mezzo di scambio per saldare i debiti all'interno dell'ecosistema TRUST.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.24;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract TrustToken is ERC20 {
    uint256 public constant RATE = 1000;

    constructor() ERC20("Trust Token", "TRUST") {}

    function mint() external payable {
        require(msg.value > 0, "TrustToken: must send ETH to mint tokens");
        uint256 amountToMint = msg.value * RATE;
        _mint(msg.sender, amountToMint);
    }
}
```

Listing 6: Contratto TrustToken.sol completo

Scelte Implementative

- **Ereditarietà da OpenZeppelin:** La scelta di ereditare dal contratto `ERC20` di OpenZeppelin è una best practice consolidata. Fornisce un'implementazione sicura, testata e standard del token, riducendo la superficie di attacco e garantendo la compatibilità con l'ecosistema Ethereum (wallet, exchange, etc.).
- **Funzione `mint()`:** La funzione `mint` è stata resa `public` e `payable` per soddisfare il requisito di permettere a chiunque di coniare nuovi token in cambio di Ether. L'uso di

`msg.value` permette di ricevere ETH, che vengono usati per calcolare la quantità di token da emettere.

- **Tasso di Cambio Fisso:** È stata definita una costante `RATE` per stabilire un tasso di cambio fisso tra ETH e TRUST. Questo semplifica la logica di conio, anche se in un'applicazione reale si potrebbe optare per un meccanismo di prezzo più dinamico (es. oracoli).

3.3 Analisi degli Script di Interazione

Per dimostrare la funzionalità e la robustezza del sistema, sono stati creati due script TypeScript distinti, ognuno con uno scopo preciso: `interaction.ts` funge da demo completa del "happy path", mentre `robustness-test.ts` funge da stress test per validare le misure di sicurezza.

Script `interaction.ts`: La Demo End-to-End Più che un test, questo script è la "sceneggiatura" di una demo completa, progettata per simulare un caso d'uso realistico e complesso e per dimostrare che tutti i componenti del sistema funzionano insieme armoniosamente. Lo script è organizzato in fasi sequenziali che coprono l'intero ciclo di vita di un caso d'uso:

1. **Setup:** Deploy dei contratti `TrustToken` e `TrustManager`.
2. **Creazione e Spese:** Creazione di un gruppo e aggiunta di diverse spese complesse usando tutte e tre le modalità.
3. **Analisi e Semplificazione:** Visualizzazione dei saldi prima e dopo la chiamata a `simplifyDebts`.
4. **Saldo Debiti:** Minting dei token, approvazione della spesa e saldo di debiti multipli.
5. **Verifica Finale:** Controllo che tutti i saldi nel manager siano a zero e che i token siano stati trasferiti correttamente.
6. **Test di Fallimento Controllato:** Esecuzione di una transazione che deve fallire per una rapida prova di robustezza.

Questo approccio strutturato rende la demo chiara e permette di apprezzare ogni funzionalità del sistema in un unico flusso logico.

Script `robustness-test.ts`: La Prova di Robustezza Se `interaction.ts` dimostra che il sistema funziona, `robustness-test.ts` dimostra che "non si rompe". Il suo unico scopo è "attaccare" deliberatamente il contratto per provare, in modo sistematico, la validità di ogni meccanismo di sicurezza e di ogni `require` statement. Lo script tenta di eseguire una serie di azioni non valide, verificando che il contratto le rifiuti correttamente. Le categorie di test includono:

- **Violazioni sulla Gestione dei Gruppi:** Tentativi di aggiungere spese da parte di non-membri o di unirsi a un gruppo due volte.
- **Violazioni sulla Logica delle Spese:** Registrazione di spese con dati inconsistenti (es. percentuali o somme errate).
- **Violazioni sulla Sicurezza dei Pagamenti:** Tentativi di saldare debiti inesistenti, di pagare non-creditori o di eseguire trasferimenti senza la necessaria approvazione del token.

Avere uno script dedicato a validare la resilienza del contratto è una best practice che ne aumenta l'affidabilità e dimostra un approccio maturo allo sviluppo di smart contract.

3.4 Analisi dei Test Unitari e di Integrazione

La fase di testing è stata condotta utilizzando il framework integrato di Hardhat, che si avvale di **Mocha** per la struttura e **Chai** per le asserzioni. Sono stati scritti test approfonditi sia per `TrustManager` che per `TrustToken`.

Strategia di Testing

- **Isolamento:** Ogni test viene eseguito in un ambiente pulito grazie all'uso del blocco `beforeEach`, che deploya una nuova istanza dei contratti prima di ogni esecuzione.
- **Copertura Completa:** I test coprono non solo il "happy path" (il funzionamento corretto), ma anche i casi limite e le condizioni di errore.
- **Revert Conditions:** Si verifica che il contratto rifiuti transazioni non valide, controllando i messaggi di errore specifici con `to.be.revertedWith(...)`.
- **Eventi:** I test verificano che gli eventi vengano emessi correttamente con i parametri attesi, usando `.to.emit(...).withArgs(...)`.

```
it("Should revert if the token transfer fails (e.g., not enough allowance)",
    async function(){
    // Bob ha bisogno di token per pagare, quindi li conia
    await token.connect(bob).mint({value: ethers.parseEther("0.1")});

    // Bob prova a saldare il debito SENZA prima approvare la spesa.
    // Ci aspettiamo che la chiamata a `transferFrom` dentro `settleDebt`
    fallisca.
    await expect(manager.connect(bob).settleDebt(0, alice.address))
        .to.be.revertedWithCustomError(token, "ERC20InsufficientAllowance");
});
```

Listing 7: Esempio di test per una condizione di revert

Questa suite di test garantisce un'elevata affidabilità del codice e ha permesso di individuare e correggere bug durante lo sviluppo.

4 Il Flusso Operativo di TRUST: dalla Spesa al Saldo

Dopo aver analizzato i dettagli tecnici dei singoli contratti, è utile fare un passo indietro per illustrare il ciclo di vita completo di un debito all'interno dell'ecosistema TRUST. Per farlo, seguiremo l'esatto e complesso scenario implementato nello script di demo `interaction.ts`, mostrando come il sistema gestisce una serie di spese incrociate per poi risolverle in modo efficiente.

4.1 Fase 1: Registrazione e Aggregazione dei Debiti Complessi

Il processo inizia con la creazione di un gruppo da parte di Alice, che include Bob, Charlie e David. Successivamente, vengono registrate quattro spese con diverse modalità di divisione, creando un grafo dei debiti volutamente ingarbugliato.

1. **Spesa 1 (Equa):** Alice paga 100 per tutti e 4. Ognuno doveva 25.
   - Saldo Alice: '+75'
   - Saldi B, C, D: '-25' ciascuno.
2. **Spesa 2 (Percentuale):** Bob paga 80 per Charlie (60%, cioè 48) e David (40%, cioè 32).
   - Saldo Bob: '-25 + 80 = +55'
   - Saldo Charlie: '-25 - 48 = -73'
   - Saldo David: '-25 - 32 = -57'
3. **Spesa 3 (Esatta):** Charlie paga 50 per Alice.
   - Saldo Charlie: '-73 + 50 = -23'
   - Saldo Alice: '+75 - 50 = +25'
4. **Spesa 4 (Esatta):** David paga 20 per Bob.
   - Saldo David: '-57 + 20 = -37'
   - Saldo Bob: '+55 - 20 = +35'

Al termine di questa fase, i saldi netti aggregati nel mapping `balances` sono i seguenti:

```
- Alice:   +25
- Bob:     +35
- Charlie: -23
- David:   -37
```

La situazione è complessa: chi deve pagare chi? David deve 37, ma a chi? E Charlie? È qui che la semplificazione diventa essenziale.

4.2 Fase 2: La Semplificazione che Trasforma il Caos in Ordine

A questo punto, un membro del gruppo (nel nostro script, Alice) invoca la funzione `simplifyDebts`. Il contratto esegue l'algoritmo greedy partendo dai saldi netti:

- **Creditori:** Bob (+35), Alice (+25)
- **Debitori:** David (-37), Charlie (-23)

L'algoritmo calcola il piano di pagamenti ottimale per risolvere questa situazione con il minor numero di transazioni. Il risultato è il seguente piano di pagamenti:

- **David** deve pagare un totale di 37, suddivisi in:
  - 25 ad Alice
  - 12 a Bob
- **Charlie** deve pagare 23 a Bob.

Una volta calcolato questo piano, la funzione **azzera e sovrascrive** i valori nel mapping `balances` per riflettere questo nuovo "grafo dei debiti" semplificato. Il mapping ora non rappresenta più un saldo netto generico, ma un vero e proprio piano di pagamenti diretto e attuabile.

4.3 Fase 3: Il Saldo Finale: un Percorso Chiaro grazie alla Logica Off-Chain

A questo punto sorge una domanda fondamentale: se il contratto, dopo la semplificazione, memorizza solo i saldi finali ('Alice: +25, Bob: +35, Charlie: -23, ...'), come fa un utente come Charlie a sapere che deve pagare specificamente Bob e non Alice?

La risposta risiede nella separazione dei compiti tra lo smart contract (on-chain) e l'applicazione utente (off-chain, es. un'interfaccia web o uno script). Questa è una best practice fondamentale nello sviluppo di DApp.

Il Ruolo della DApp (Logica Off-Chain) Lo smart contract agisce come unica fonte di verità per i dati, ma è compito dell'applicazione che l'utente utilizza interpretare questi dati e presentarli in modo leggibile. L'interfaccia utente (DApp) esegue i seguenti passaggi:

1. **Legge i saldi semplificati** dal mapping `balances` del contratto.
2. **Esegue a sua volta l'algoritmo greedy off-chain**. Poiché questa operazione avviene nel browser o sul computer dell'utente (in JavaScript/TypeScript), è istantanea e a costo zero.
3. L'output di questo calcolo off-chain è il **piano di pagamenti esatto e non ambiguo**, che viene poi mostrato all'utente.

L'Esperienza Utente Concreta Grazie a questa logica off-chain, il percorso per l'utente diventa chiaro e guidato:

1. **Charlie** apre la DApp. L'applicazione legge i saldi, esegue l'algoritmo e gli mostra un unico messaggio chiaro: "Devi pagare 23 a Bob", con un pulsante "Paga" che pre-compila la transazione corretta. A questo punto, Charlie esegue la transazione:
   - Chiama `settleDebt(0, bob.address)` per trasferire 23 token. Il suo saldo on-chain va a 0.
2. **David** apre la DApp e vede un'interfaccia simile, che gli presenta due azioni distinte: "Paga 12 a Bob" e "Paga 25 ad Alice". Esegue le due transazioni:
   - Chiama `settleDebt(0, bob.address)` per trasferire 12 token a Bob.
   - Chiama `settleDebt(0, alice.address)` per trasferire i restanti 25 token ad Alice. Il suo saldo on-chain va a 0.

Al termine di queste operazioni, tutti i saldi nel `balances` del gruppo sono tornati a zero. Mantenere on-chain solo i dati essenziali e delegare la logica di presentazione all'applicazione off-chain non solo fa risparmiare gas, ma è anche il segreto per creare un'esperienza utente fluida e intuitiva.

5 Analisi Approfondita delle Vulnerabilità e Contromisure

Una valutazione rigorosa della sicurezza è essenziale per qualsiasi smart contract. Di seguito, un'analisi dettagliata dei rischi e delle contromisure adottate.

5.1 Re-entrancy

Vettore di Attacco Questa è una delle vulnerabilità più note e pericolose. Un attacco di re-entrancy si verifica quando un contratto esterno malevolo, chiamato da una nostra funzione, è in grado di richiamare la stessa funzione (o un'altra funzione del nostro contratto) prima che la prima invocazione sia terminata. Nella nostra DApp, la funzione `settleDebt` è un potenziale bersaglio, poiché chiama la funzione `transferFrom` di un contratto esterno (il token ERC-20). Se un token malevolo fosse usato, il suo `transferFrom` potrebbe contenere una logica per richiamare `settleDebt`, potenzialmente prosciugando i fondi dei creditori prima che i saldi vengano aggiornati correttamente.

Mitigazione: Pattern "Checks-Effects-Interactions" Per prevenire categoricamente questa vulnerabilità, la funzione `settleDebt` è stata implementata seguendo scrupolosamente il pattern "Checks-Effects-Interactions".

1. **Checks:** Vengono eseguiti tutti i controlli di validità all'inizio della funzione (es. il pagatore ha un saldo negativo, il ricevente uno positivo).
2. **Effects:** Lo stato del nostro contratto (`balances`) viene aggiornato **immediatamente dopo** i controlli. Questo è il passo cruciale: anche se un attacco di re-entrancy avvenisse, lo stato del nostro contratto sarebbe già consistente.
3. **Interactions:** La chiamata al contratto esterno (`token.transferFrom(...)`) è l'ultima operazione eseguita.

Questo pattern garantisce che, al momento di qualsiasi interazione esterna, lo stato interno del contratto sia già stato aggiornato in modo atomico, rendendo inefficace qualsiasi tentativo di rientro.

5.2 Gestione degli Errori e Validazione degli Input

Vettore di Attacco Input malformati o imprevisti possono portare a stati inconsistenti del contratto. Ad esempio, un utente potrebbe tentare di aggiungere una spesa in un gruppo di cui non è membro, o fornire array di lunghezze diverse per debitori e importi.

Mitigazione: Uso Estensivo di `require` Ogni funzione pubblica che modifica lo stato inizia con un blocco di istruzioni `require` che validano rigorosamente tutti gli input e le pre-condizioni necessarie.

- **Controllo di Appartenenza:** Funzioni come `addExpense...` e `settleDebt` verificano che `msg.sender` sia un membro del gruppo specificato.
- **Consistenza dei Dati:** Viene controllato che gli array di input abbiano lunghezze corrispondenti e che la somma delle quote corrisponda al totale della spesa.
- **Prevenzione di Stati Logici Impossibili:** Si impedisce a un utente di pagare un debito se non ne ha, o di pagare un utente che non è un creditore.

Questo approccio di "fail-fast" garantisce che qualsiasi transazione non valida venga annullata immediatamente, preservando l'integrità dello stato.

Nota sull'Efficienza: Custom Errors A partire dalla versione 0.8.4 di Solidity, è stata introdotta una feature chiamata *Custom Errors*. Invece di usare stringhe di testo nei 'require' (es. 'require(condizione, "Messaggio di errore")'), si possono definire errori personalizzati a livello di contratto:

```
// Definizione dell'errore
error CallerNotAMember(address user, uint256 groupId);

// Utilizzo nella funzione
if (!_isMemberOf(_groupId, msg.sender)) {
    revert CallerNotAMember(msg.sender, _groupId);
}
```

Questo approccio non solo rende il codice più pulito e leggibile, ma è anche significativamente **più efficiente in termini di gas**, sia in fase di deploy del contratto che durante l'esecuzione (in caso di revert), poiché gli identificatori degli errori sono più compatti delle stringhe. Per una DApp ottimizzata per la produzione, l'uso dei Custom Errors è considerato una best practice.

5.3 Vulnerabilità Aritmetiche (Overflow/Underflow)

Vettore di Attacco Le versioni precedenti di Solidity erano suscettibili a errori di overflow e underflow aritmetico.

Mitigazione: Compilatore Moderno Il progetto utilizza una versione di Solidity (`^0.8.24`) che, a partire dalla versione 0.8.0, include controlli nativi per queste condizioni. Qualsiasi operazione che causerebbe un overflow o un underflow lancia automaticamente un errore e annulla la transazione. Questo elimina la necessità di utilizzare librerie esterne come *SafeMath*.

5.4 Denial of Service (DoS) tramite Limiti di Gas

Vettore di Attacco Un utente malintenzionato potrebbe creare uno scenario in cui l'esecuzione di una funzione richiede una quantità di gas superiore al limite del blocco, rendendo di fatto quella funzione inutilizzabile.

Analisi del Rischio nel Progetto Il principale vettore di questo tipo di attacco nel nostro contratto è la funzione `simplifyDebts`, a causa dell'algoritmo di ordinamento `_sortBalances` che ha una complessità temporale di $O(N^2)$. Se un utente creasse un gruppo con un numero estremamente elevato di membri, la chiamata a `simplifyDebts` potrebbe fallire per esaurimento del gas.

Mitigazione e Compromessi Per questo progetto, il rischio è stato considerato accettabile, poiché l'attacco ha un costo per l'attaccante e non porta a una perdita diretta di fondi. In un'applicazione di produzione su larga scala, tuttavia, questo rischio andrebbe mitigato. La soluzione standard per questo tipo di problema è **spostare il calcolo pesante off-chain**. Anziché eseguire l'ordinamento on-chain, il frontend dell'applicazione (DApp) leggerebbe i saldi, eseguirebbe l'algoritmo di ordinamento $O(N^2)$ localmente e passerebbe le liste di debitori e creditori già ordinate alla funzione smart contract. La funzione on-chain, a quel punto, dovrebbe solo *verificare* che le liste siano ordinate, un'operazione molto più economica con complessità $O(N)$, prima di procedere con il matching. Questo pattern massimizza l'efficienza del gas delegando il lavoro computazionale al client.

5.5 Timestamp Dependence

Vettore di Attacco L'uso di `block.timestamp` può introdurre una vulnerabilità se la logica del contratto dipende da un tempo preciso, poiché i miner possono manipolare il timestamp di un blocco entro una finestra di alcuni secondi.

Analisi del Rischio nel Progetto Nel nostro contratto, `block.timestamp` è usato solo per registrare il momento di una spesa. Poiché questa è un'informazione a scopo di registrazione e non guida alcuna logica critica, una piccola imprecisione è considerata innocua e non costituisce una vulnerabilità pratica per il nostro caso d'uso.

6 Valutazione del Costo del Gas

L'efficienza è un parametro fondamentale per la usability di una DApp, dato che ogni operazione on-chain ha un costo reale per l'utente. Per quantificare le performance del sistema TRUST, è stata condotta un'analisi approfondita dei costi di gas utilizzando il plugin `hardhat-gas-reporter`. Questo strumento, eseguito durante la suite di test, calcola il consumo di gas per ogni funzione del contratto e, se configurato, può stimare il costo monetario equivalente.

Per fornire una stima realistica, il plugin è stato configurato per interfacciarsi con due servizi esterni tramite le loro rispettive API, come visibile nel file `hardhat.config.ts`:

- **Etherscan API:** Viene utilizzata per interrogare la rete Ethereum e ottenere il prezzo medio del gas in tempo reale (espresso in Gwei). Questo assicura che il costo della transazione sia basato su condizioni di mercato attuali.
- **CoinMarketCap API:** Viene usata per ottenere il tasso di cambio istantaneo tra ETH e USD.

La combinazione di questi due dati permette di tradurre il consumo di gas astratto (un'unità di calcolo) in un costo monetario concreto e attuale, offrendo una prospettiva tangibile sull'efficienza economica del contratto.

La tabella seguente riassume i dati raccolti, mostrando i costi di esecuzione minimi, massimi e medi, il numero di chiamate effettuate durante i test e il costo medio stimato in USD per ogni operazione.

Tabella 1: Report dettagliato del consumo di gas e dei costi stimati.

Contratto	Funzione	Gas Min	Gas Max	Avg. Gas	Costo (USD)
TrustManager					
	`createGroup`	141.634	238.373	215.026	$2.59
	`joinGroup`	-	-	82.679	$0.99
	`addExpenseEqually`	183.129	231.114	208.458	$2.51
	`addExpenseByPercentage`	-	-	230.921	$2.78
	`addExpenseWithExactAmounts`	161.786	227.625	193.468	$2.33
	`settleDebt`	-	-	61.971	$0.75
	`simplifyDebts`	42.565	49.889	**45.006**	**$0.54**
TrustToken					
	`approve`	46.952	46.964	46.961	$0.57
	`mint`	-	-	68.201	$0.82
	`transfer`	-	-	52.232	$0.63
	`transferFrom`	-	-	58.486	$0.70
Costo di Deployment					
	`TrustManager`	-	-	3.281.799	$39.49
	`TrustToken`	-	-	1.036.656	$12.47

Analisi dei Risultati Dall'analisi della tabella emergono diverse considerazioni chiave sulle scelte di progettazione:

- **Costo delle Funzioni di Spesa:** Le funzioni `addExpense...` sono, come previsto, tra le più costose in termini di gas. Questo è dovuto al fatto che, per ogni spesa, il contratto deve eseguire molteplici operazioni di scrittura sullo storage (`SSTORE`), una per ogni saldo utente da aggiornare. Poiché `SSTORE` è una delle istruzioni più costose della EVM, il costo è direttamente proporzionale al numero di partecipanti alla spesa.

- **L'Efficienza di `simplifyDebts`:** Il risultato più significativo è il costo sorprendentemente basso della funzione `simplifyDebts`, con una media di soli **45.006 gas**. Questo dato è la prova diretta del successo della scelta architetturale di mantenere i saldi netti aggiornati incrementalmente. Se avessimo dovuto calcolare i saldi da un grafo dei debiti all'interno di questa funzione, il costo sarebbe stato di un ordine di grandezza superiore, rendendo la DApp inutilizzabile. Invece, grazie a questa ottimizzazione, la semplificazione è l'operazione più economica del contratto.
- **Costo Moderato del Saldo:** La funzione `settleDebt` ha un costo moderato. Questo costo include la lettura di due saldi, la scrittura di due saldi e l'interazione esterna con il contratto del token tramite `transferFrom`, che a sua volta ha un costo. È un costo ragionevole per un'operazione che finalizza un pagamento on-chain.
- **Costi di Deployment:** I costi di deploy dei contratti sono elevati, specialmente per il `TrustManager` che contiene molta logica di business. Questo è normale per smart contract di una certa complessità ed è un costo che viene pagato una sola volta al momento della creazione del sistema sulla blockchain.

In sintesi, l'analisi del gas conferma che le scelte architetturali, in particolare il refactoring della gestione dei saldi, sono state efficaci nel rendere il sistema economicamente sostenibile, bilanciando il costo delle operazioni di scrittura con l'efficienza delle operazioni di lettura e calcolo.

7 Manuale d'Uso

Questa sezione fornisce le istruzioni per installare, testare e interagire con il progetto TRUST in un ambiente di sviluppo locale.

7.1 Prerequisiti

- Node.js (versione 18 o superiore)
- npm (Node Package Manager) o yarn

7.2 Installazione

Clonare la repository del progetto e installare le dipendenze:

```
git clone <URL_DELLA_REPOSITORY>
cd <NOME_CARTELLA_PROGETTO>
npm install
```

7.3 Compilazione dei Contratti

Per compilare i contratti Solidity e generare gli artifact e i type bindings per TypeScript, eseguire:

```
npx hardhat compile
```

Questo creerà una cartella `/artifacts` e `/typechain-types`.

7.4 Esecuzione dei Test

Per eseguire la suite completa di test unitari e di integrazione, usare il comando:

```
npx hardhat test
```

Verrà mostrato un output dettagliato con l'esito di ogni test. Per generare anche il report sul gas, i test vengono eseguiti automaticamente con il reporter configurato.

7.5 Esecuzione degli Script di Interazione

Per interagire con il progetto e vederne le funzionalità in azione, è necessario prima avviare un nodo Hardhat locale, che simula la blockchain di Ethereum sul proprio computer.

1. Avviare il Nodo Locale In un terminale, eseguire il seguente comando. Questo avvierà una blockchain di test e fornirà 20 account prefinanziati con 10000 ETH di prova ciascuno. **Lasciare questo terminale aperto** durante l'esecuzione degli script.

```
npx hardhat node
```

2. Eseguire gli Script In un **secondo terminale**, è possibile lanciare uno dei due script di interazione.

Demo Principale: Per eseguire la demo completa che mostra un flusso di utilizzo realistico dall'inizio alla fine:

```
npx hardhat run scripts/interaction.ts --network localhost
```

L'output mostrerà passo dopo passo tutte le operazioni eseguite: la creazione del gruppo, l'aggiunta delle spese, i saldi prima e dopo la semplificazione, e la verifica finale.

Test di Robustezza: Per verificare la resilienza del contratto e la corretta gestione degli errori:

```
npx hardhat run scripts/robustness-test.ts --network localhost
```

L'output mostrerà una serie di test, ognuno dei quali confermerà che un'azione non valida è stata correttamente bloccata dal contratto, provandone la sicurezza.