CS190: Blockchain Programming and Applications

Lecture 6: Smart Contract Design Patterns

Yanju Chen

- HW1 is due next Monday (Oct 20)
- Slides and code of this lecture is available on course website.

UCSB

Why Design Patterns Matter?

There are blockchain incidents causing big losses…

The DAO (2016)

A reentrancy bug let an attacker recursively withdraw funds before balances updated, stealing about $50–60M.

Parity Multisig (2017)

A library/delegatecall initialization flaw allowed takeover/locking of multisig libraries, affecting about 153,000 ETH (~$30–34M).

bZx (2020)

Flash loans were used to manipulate on-chain prices and create profitable trades, costing the protocol millions (≈$8M in a major attack).

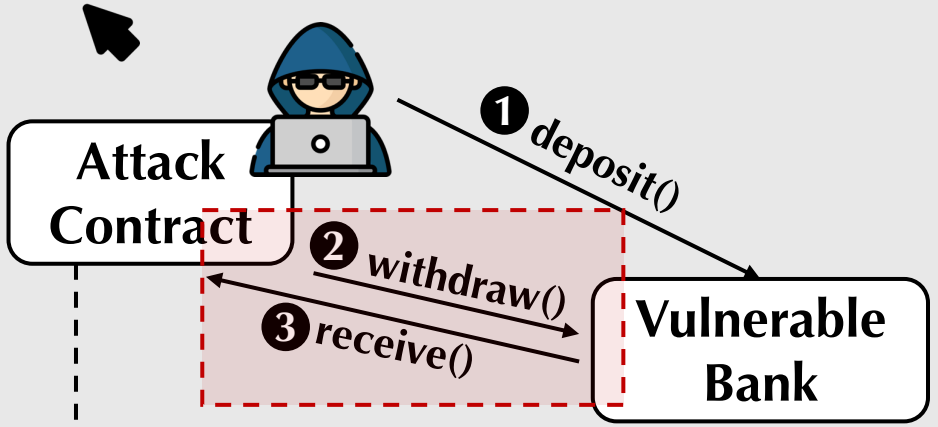
Harvest Finance (Oct 2020)

Large, fast trades distorted AMM prices via flash loans and drained vault liquidity, losing roughly $24M.

Design patterns can make intent explicit, reduce irreversible mistakes, improve auditability, contain risk, clarify legal states, ensure liveness & conservation, enable composition & reuse, …

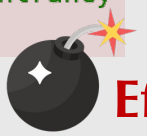
Reentrancy Attacks in Solidity DEMO

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

contract VulnerableBank {
    mapping(address => uint256) public balanceOf;

    /// @notice Deposit ETH into sender's balance.
    function deposit() external payable {
        require(msg.value > 0, "no ether");
        balanceOf[msg.sender] += msg.value;
    }

    /// @notice Withdraw entire sender balance — vulnerable ordering.
    function withdraw() external {
        uint256 bal = balanceOf[msg.sender];
        require(bal > 0, "no balance");

        // Vulnerable: external call before state update
        (bool ok, ) = msg.sender.call{value: bal}("");
        require(ok, "send failed");

        // State update happens after the external call — allow reentrancy
        balanceOf[msg.sender] = 0;
    }
}
```



"re-enter" multiple times

```solidity
/// ...
/// @notice Receive callback used to reenter bank.withdraw()
receive() external payable {
    reenterTimes++;

    // Continue reentering while bank has at least the originally
    // deposited amount and we haven't exceeded hops.
    if (address(bank).balance >= msg.value && _hop < maxHops) {
        _hop++;
        bank.withdraw();
    }
}
```



Effects after Interactions

Reentrancy: Preventative Techniques DEMO

```
// Vulnerable: external call before state update
(bool ok, ) = msg.sender.call{value: bal}("");
require(ok, "send failed");

// State update happens after the external call — allow reentrancy
balanceOf[msg.sender] = 0;
```

Effects after Interactions

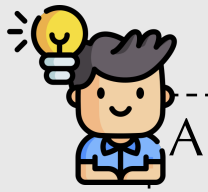
```
// State update happens before the external call — prevent reentrancy
balanceOf[msg.sender] = 0;

// Vulnerable: external call before state update
(bool ok, ) = msg.sender.call{value: bal}("");
require(ok, "send failed");
```

Checks-Effects-Interactions

```
bool internal locked;
modifier noReentrant() {
    require(!locked, "No re-entrancy");
    locked = true;
    _;
    locked = false;
}
```

Reentrancy Guard

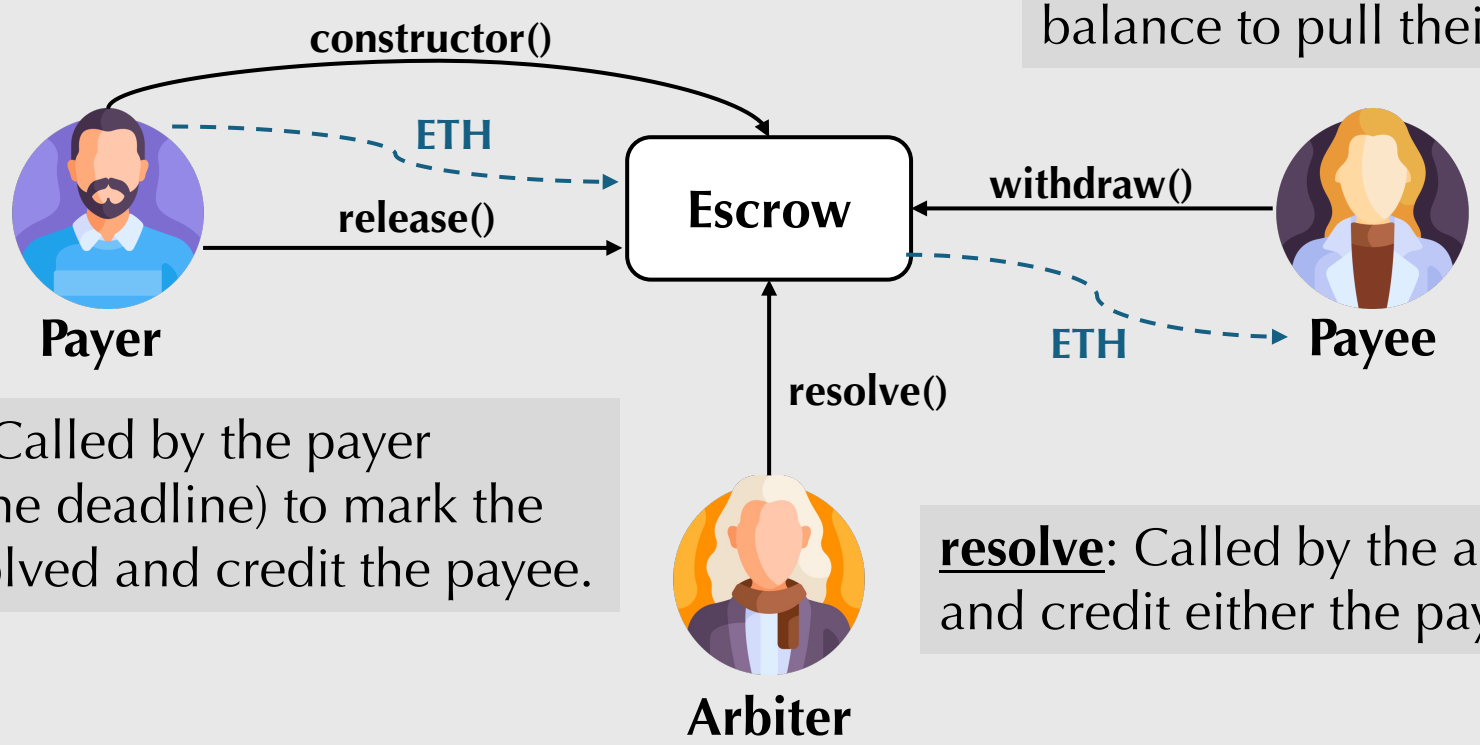
Example #1: Escrow



An **escrow** holds a **payer**'s funds until they're released to the **payee** or decided by an **arbiter**.

constructor: Initializes the contract with payer, payee, arbiter, a deadline, and funds the escrow.

withdraw: Allows an address with a credited balance to pull their funds from the contract.



release: Called by the payer (before the deadline) to mark the deal resolved and credit the payee.

resolve: Called by the arbiter to choose a winner and credit either the payee or the payer.

Design Patterns in Escrow Contract DEMO

Pull-over-Push

Credit winners; they then pull funds via withdraw().

Checks–Effects–Interactions

Do checks and state changes before any external call.

Single Withdrawal Path

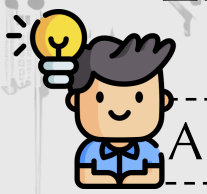
Route all transfers through one withdraw() function.

```
function withdraw() external {
    uint256 due = credit[msg.sender];
    if (due == 0) revert NothingToWithdraw();
    (bool ok, ) = msg.sender.call{value: due}("");
    credit[msg.sender] = 0;
    require(ok, "transfer failed");
    emit Withdrawn(msg.sender, due);
}
```

Effects after Interactions

Can we trigger a reentrancy attack here?

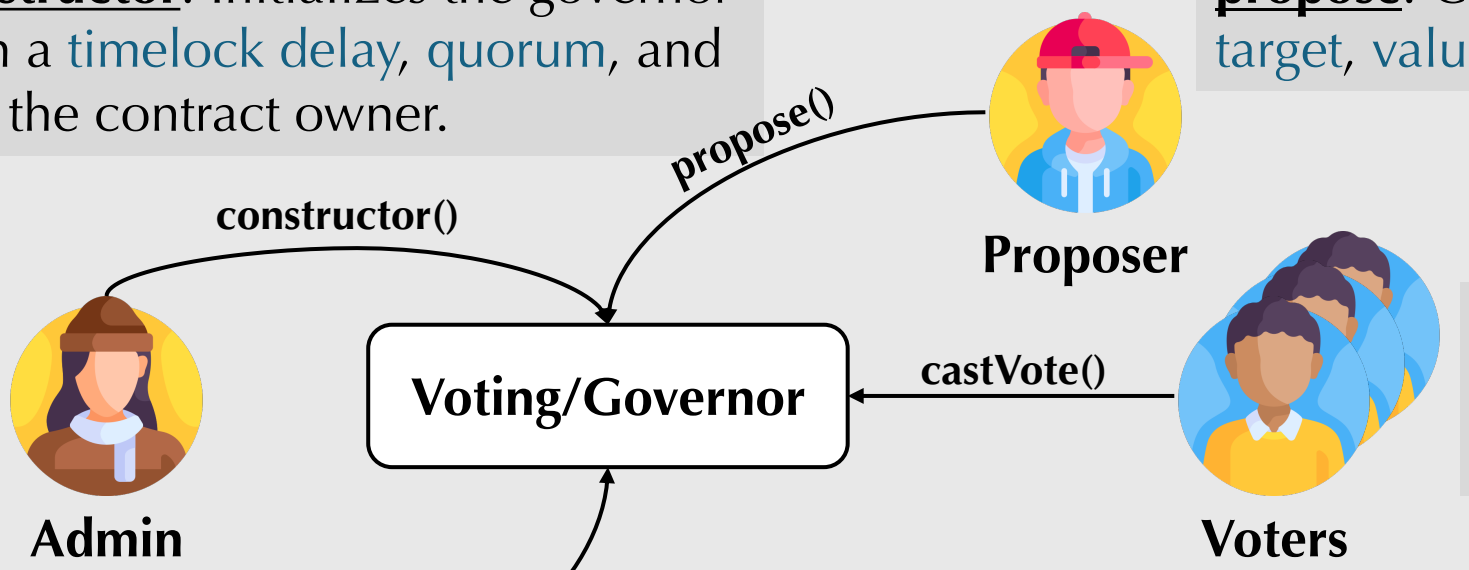
Example #2: Voting/Governor



A **Governor** lets **proposers** suggest, **voters** decide, and approved actions execute after a timelock.

constructor: Initializes the governor with a timelock delay, quorum, and sets the contract owner.

propose: Creates a new proposal defining the target, value, calldata, and voting window.



castVote: Records a voter's support or opposition using snapshot voting power from a past block.

queue: Moves a passed proposal into the timelock queue and sets its earliest execution time.

execute: Executes the approved and queued proposal after the delay has expired.

Design Patterns in Voting/Governor Contract DEMO

Snapshot Voting

Fix voting power at a past block to stop manipulation.

```
/// @notice Execute after the timelock: calls exactly
/// the recorded payload to an allowlisted target.
function execute(bytes32 id) external {
    Proposal storage p = _require(id);
    require(p.queued, "not queued");
    require(!p.executed, "executed");
    require(block.timestamp >= eta[id], "too early");
    require(allowedTarget[p.target], "target not allowed");

    p.executed = true;
    (bool ok, ) = p.target.call{value: p.value}(p.callData);
    require(ok, "exec failed");
    emit Executed(id);
}
```

Timelock Delay

Wait after passing so users can review/prepare.

Whitelisted Execution

Only call pre-approved targets.

Auditability

Every step is on-chain and verifiable.

Event-Driven, Off-Chain Trigger

Bots/users listen to events and trigger execution.

Check out an example contract!

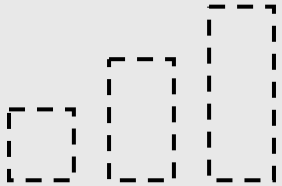
Example #3: Commit-Reveal Auction

What is a **commit-reveal auction**?

A commit–reveal auction lets bidders first **hide** bids with a hash, then **reveal** them later for fair comparison.

Phases **Commit** ⟶ **Reveal** ⟶ **Finalized**

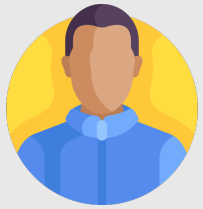
Alice



deposit() * 3

actual bid
15% deposit

reveal()



Bob

deposit() * 2

actual bid
100% deposit

reveal()

Winner

Charlie

deposit() * 2

did not reveal;
therefore no bidding

Example #3: Commit-Reveal Auction

What is a **commit-reveal auction**?

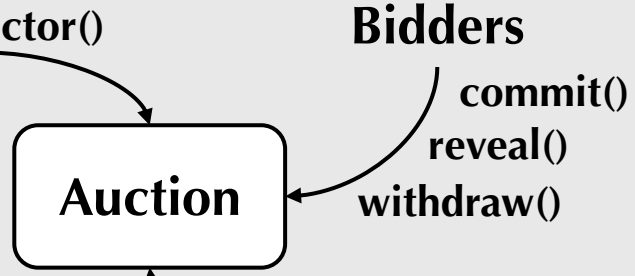
A commit–reveal auction lets bidders first **hide** bids with a hash, then **reveal** them later for fair comparison.

constructor: Initializes the auction by setting the seller, phase, and time deadlines for commit and reveal periods.

Bidders

Seller

Winner



commit: Lets bidders submit a hashed bid with a deposit during the commit phase.

reveal: Allows bidders to reveal their real bid and salt for verification, updating the highest bid if valid.

withdrawal: Handles all payouts after the auction ends: seller claims payment, winner gets change, others get refunds.

Design Patterns in Auction Contract DEMO

State Machine with Time-Based Transitions

Enforces Commit → Reveal → Finalized by timestamps so actions only run in the right phase.

```
/// @notice Reveal the real bid and salt; deposit must cover the bid
function reveal(uint256 bid, bytes32 salt) external inPhase(Phase.Reveal) {
    bytes32 c = commitments[msg.sender];
    if (c == 0) revert NoCommitment();
    if (keccak256(abi.encode(bid, salt)) != c) revert BadReveal();
    if (deposits[msg.sender] < bid) revert BadReveal(); // deposit not enough

    // record highest bid
    if (bid > highestBid) {
        highestBid = bid;
        highestBidder = msg.sender;
    }

    // prevent reusing the same commitment
    commitments[msg.sender] = 0;
}
```

Commit-Reveal

Commit a hidden bid hash, reveal later to prove it—secrecy first, verification later.

Check out an example contract!

One-Time Commitment

Clear the commitment after reveal to prevent double reveals or reuse.