#6 Smart Contract Design Patterns

Lecture Notes for CS190N: Blockchain Technologies and Security — October 15, 2025

In this lecture, we build a practical vocabulary of smart contract design patterns for adversarial and immutable environments. First, we motivate why patterns matter and frame each one by intent, participants, and invariants that anchor correctness. Next, we study three core patterns through compact Solidity fragments: Escrow for conditional value transfer using pull-over-push payouts and timeouts, Voting for collective decisions using snapshot weighting, quorums and thresholds with timelocked execution, and State Machines for phase-based protocols with explicit guards and commit–reveal workflows. Then, we examine typical failure modes and show how Checks–Effects–Interactions and restricted external calls keep outcomes predictable and funds retrievable. Finally, we connect the patterns in a short end-to-end example where a DAO selects a vendor via auction, settles through escrow, and updates parameters through governance. By the end, you will be able to read and write minimal yet safe contracts that make outcomes explicit, route value transfers through a single withdrawal path, and reason about correctness through well-stated invariants.

1 INTRODUCTION: WHY PATTERNS IN SMART CONTRACTS?

Smart contracts make design choices both irreversible and public. After deployment, code is difficult to change, every path is observable, and each external call crosses a trust boundary. In this environment a design pattern is more than style. It packages a recurring problem together with roles and invariants that must hold, so that value moves in predictable ways and reviewers focus on the right places.

This chapter develops three patterns that recur across applications. *Escrow* governs conditional value transfer. *Voting* produces auditable collective change using a *snapshot* of voting power and a scheduled execution through a *timelock*. *State machines* encode phase based protocols so that functions are callable only in valid states. The quick map in Table 1 will guide what follows. When a new term appears, it is defined inline. A snapshot is the voting power of an account measured at a past block. A timelock is a delay that must elapse between a successful vote and execution. Commit and reveal means first committing to a hash of a value and later revealing the value and a secret salt that reproduces the hash.

Table 1. Recurring problems and suitable patterns

Problem theme	**Representative pattern(s)**
Conditional value transfer	Escrow; pull over push payouts
Collective, auditable change	Voting with snapshot; timelock
Phase based interaction	State machine; commit and reveal

2 ESCROW

Imagine a client deposits 5 ETH to pay a developer. Payment should be released if the client accepts delivery. Otherwise the client should recover the deposit after a deadline or by an arbiter's decision. A workable escrow enforces three ideas at all times.

(1) The contract preserves the full amount that was deposited. This is conservation of value.
(2) The deal ends in exactly one outcome. Either release to the payee or refund to the payer.
(3) Someone can always withdraw eventually. A timeout or an arbiter ensures liveness.

For a single deal a very small data layout is enough. We keep the parties, the deposited amount, a `deadline`, a two value state, and a ledger that records who may later pull funds. The ledger is the key. Instead of sending ETH during state transitions, the contract credits one party and lets them withdraw. This separation keeps the critical logic free of external calls. See Table 2 for a compact summary, then we move to a concrete example.

Table 2. Minimal single deal escrow layout

Element	Sketch
Parties	`payer`, `payee`, optional `arbiter`
Deal state	`enum {Funded, Resolved}`
Timeout	`deadline` (Unix timestamp)
Payout ledger	`mapping(address=>uint256) credit`
Core actions	`release()`, `refund()`, `resolve(bool)`, `withdraw()`

Example 1 (Escrow core fragments). The next fragment shows only the transition and payout logic. Treat it as rules that must hold before and after each call. Boilerplate such as events is omitted so the core is visible.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract EscrowExample {
    enum State { Funded, Resolved }
    mapping(address => uint256) credit;
    address payer; address payee; address arbiter;
    uint256 amount; uint256 deadline; State s;

    error NotPayer(); error NotArbiter(); error NothingToWithdraw();

    modifier onlyPayer() { if (msg.sender != payer) revert NotPayer(); _; }
    modifier onlyArbiter() { if (msg.sender != arbiter) revert NotArbiter(); _; }

    function release() external onlyPayer {
        require(s == State.Funded && block.timestamp < deadline, "bad state");
        s = State.Resolved; // effects first
        credit[payee] += amount; // choose exactly one winner
    }

    function refund() external onlyPayer {
        require(s == State.Funded && block.timestamp >= deadline, "bad state");
        s = State.Resolved;
        credit[payer] += amount;
    }

    function resolve(bool toPayee) external onlyArbiter {
        require(s == State.Funded, "bad state");
        s = State.Resolved;
        credit[toPayee ? payee : payer] += amount;
    }
```

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

Understanding **Example 1** is easier if we follow the causal chain and check each invariant.

(1) A terminal call changes the state to `Resolved` before any external interaction. This removes the opportunity for a receiver to reenter and affect the outcome.
(2) A terminal call records an entitlement in `credit` and does not send ETH. The real transfer happens later in `withdraw()`. If the receiver's wallet reverts, the entitlement remains and can be claimed later. The outcome does not depend on the receiver's behavior.
(3) Exactly one ledger entry increases by exactly `amount`. This preserves conservation of value and enforces the single outcome rule with one assignment.

A short trace fixes the flow. At deployment the escrow holds 5 ETH with `s = Funded`. If the client approves before the deadline then `release()` credits the payee with 5. If time runs out then `refund()` credits the payer with 5. If there is a dispute then `resolve(true)` or `resolve(false)` credits exactly one side. In every branch the ledger accounts for the full 5 ETH and only `withdraw()` touches the outside world.

An error that appears often is to send ETH directly inside `release()` or `refund()`. If that external call fails or reenters then the deal may end half finished. The present design avoids that risk by separating state mutation from interaction.

3 VOTING (GOVERNANCE)

Consider a community that wants to change a platform fee from 2% to 3%. Two properties make the system trustworthy. Voting power should be fixed before voting starts so last minute borrowing cannot sway the result. A successful decision should wait before it takes effect so everyone can see it coming. The adjustable parts that appear in practice are summarized in Table 3. The table is a checklist to consult while reading the example.

Table 3. Typical governance dimensions

Dimension	Typical choice
Eligibility and weight	Token holders; weight from snapshot block
Success rule	Quorum and majority threshold
Execution guard	Timelock delay; allowlisted targets; plain `call`

To make these properties concrete we compare two ways to measure voting power in Table 4. The snapshot approach is the safer default.

Example 2 (Governor fragments). The next two fragments encode the heart of a minimal governor. The first fragment reads voting weight at a snapshot block. The second fragment schedules and executes an allowlisted action after a delay.

```
// Snapshot based voting
```

Table 4. Reading voter weight: snapshot versus current balance

Method	Implication
Current balance at vote time	Vulnerable to temporary borrowing that is repaid after voting.
Past balance at snapshot block	Weight is fixed in advance and is not affected by later transfers.

```
interface IVotes {
    function getPastVotes(address who, uint256 blockNumber) external view returns (uint256);
}

struct Proposal {
    address target; bytes callData;
    uint256 startBlock; uint256 endBlock;
    uint256 forVotes; uint256 againstVotes;
    mapping(address => bool) voted;
}

function castVote(Proposal storage p, IVotes token, bool support) internal {
    require(block.number >= p.startBlock && block.number <= p.endBlock, "not active");
    require(!p.voted[msg.sender], "already voted");
    p.voted[msg.sender] = true;

    uint256 w = token.getPastVotes(msg.sender, p.startBlock);
    if (support) p.forVotes += w; else p.againstVotes += w;
}
```

```
// Timelocked execution with an allowlist
mapping(address => bool) allowedTarget;
uint256 minDelay;
mapping(bytes32 => uint256) eta; // proposal id -> earliest execution time

function queue(bytes32 id, Proposal storage p, uint256 quorum) internal {
    require(p.forVotes > p.againstVotes, "failed");
    require(p.forVotes >= quorum, "no quorum");
    eta[id] = block.timestamp + minDelay;
}

function execute(bytes32 id, Proposal storage p) internal {
    require(block.timestamp >= eta[id], "too early");
    require(allowedTarget[p.target], "target not allowed");
    (bool ok, ) = p.target.call(p.callData);
    require(ok, "execution failed");
}
```

Reading **Example 2** from start to finish reveals the intended control. A proposal records exactly what will happen as a target and calldata. During the voting window, each address can vote once and the weight comes from a past block. If the proposal passes thresholds then it is queued with a visible earliest execution time. After the delay expires the governor calls only allowlisted targets with the recorded calldata. This sequence is easy to audit and it removes two common sources of surprise.

A short timeline illustrates the mechanics. Suppose `startBlock = 1,200,000` and `endBlock = 1,200,960`. If `forVotes` meets quorum and exceeds `againstVotes` then the proposal is queued with `eta = now + 48h`. Everyone can inspect the target and calldata during the delay. After the delay the proposal executes exactly as written.

Two clarifications help students who build their first governor.

(1) Reading current balances looks simpler but it invites short lived loans that are repaid after voting. The snapshot severs that path.
(2) Immediate execution looks convenient but it prevents users from reacting to a change they do not want. A timelock turns surprises into scheduled actions.

4 STATE MACHINES

Some protocols must proceed through phases in order. Bids should come before reveals and settlement should come last. Encoding the phases as program state prevents premature actions, clarifies which functions are legal when, and makes time explicit in the logic. The phase and action map in Table 5 is a reading aid for the example that follows.

Table 5. Allowed actions by phase in a sealed bid auction

Phase	Actions
Commit	`commit(hash)` with deposit; no reveals accepted
Reveal	`reveal(bid, salt)`; verify hash and deposit; update leader
Finalized	Reconcile deposits; enable withdrawals; no new bids or reveals

Example 3 (Sealed bid auction fragments). We first encode the phases and the gate that restricts access. The code advances the phase when time bounds are crossed and requires the correct phase for each function.

```
enum Phase { Commit, Reveal, Finalized }
Phase phase;
uint256 commitEnd; // timestamp
uint256 revealEnd; // timestamp

modifier inPhase(Phase p) { require(phase == p, "wrong phase"); _; }

function advance() internal {
    if (phase == Phase.Commit && block.timestamp >= commitEnd) phase = Phase.Reveal;
    if (phase == Phase.Reveal && block.timestamp >= revealEnd) phase = Phase.Finalized;
}
```

The commit step binds a bidder to a number that will be revealed later. The reveal step proves that the earlier commitment matches the number and a secret salt. The salt prevents simple brute force recovery of small bids.

```
mapping(address => bytes32) commitment;
mapping(address => uint256) deposit;
address winner; uint256 best;

function commit(bytes32 c) external payable inPhase(Phase.Commit) {
    require(commitment[msg.sender] == bytes32(0), "duplicate");
    commitment[msg.sender] = c;
```

```
    deposit[msg.sender] = msg.value; // escrow funds now
}

function reveal(uint256 bid, bytes32 salt) external inPhase(Phase.Reveal) {
    require(keccak256(abi.encodePacked(bid, salt)) == commitment[msg.sender], "bad reveal");
    require(deposit[msg.sender] >= bid, "insufficient escrow");
    if (bid > best) { best = bid; winner = msg.sender; }
}
```

Transfers happen later in one place so that state transitions remain simple and predictable.

```
function withdraw() external {
    uint256 due = (msg.sender == winner) ? (deposit[msg.sender] - best) : deposit[msg.sender];
    require(due > 0, "nothing");
    if (msg.sender == winner) deposit[msg.sender] -= due; else deposit[msg.sender] = 0;
    (bool ok, ) = msg.sender.call{value: due}("");
    require(ok, "transfer failed");
}
```

Reading **Example 3** with concrete numbers anchors the ideas. Alice commits a hash of (3 ETH, `saltA`) with a 3 ETH deposit. Bob commits (5 ETH, `saltB`) with 5 ETH. During Reveal the contract recomputes the hash from (bid, salt) and checks that the deposit covers the bid. Since $5 > 3$, Bob becomes the leader with `best = 5`. After Finalized, Alice withdraws 3 ETH. Bob withdraws any excess over 5 if he deposited more. Because transfers are routed through `withdraw()`, a misbehaving receiver cannot derail the logic.

Three mistakes are worth calling out.

(1) Allowing reveals during the commit phase leaks information and lets late bidders adapt. The `inPhase` gate prevents this.
(2) Accepting commitments without a deposit invites non payable winners. The deposit ensures the winner can pay.
(3) Making transfers during state transitions mixes concerns. Keeping transfers in one function after the state is settled limits reentrancy risk and simplifies reasoning.

5 INTEGRATED EXAMPLE

This section shows how the three patterns compose into a compact flow that a student can reproduce. Vendors are selected with a sealed bid auction, payment is protected by escrow, and platform fees are governed by voting with a snapshot and a timelock. The quick checklist in Table 6 aligns each module with what must already be fixed on chain before it runs.

Table 6. Integration snapshot: what must be fixed before each step

Module	Precondition fixed on chain
Auction (state machine)	Phases and deadlines; bids escrowed; see Table 5.
Escrow	Parties, amount, deadline, arbiter set to timelock; pull withdrawal path.
Governance	Snapshot block, quorum and threshold, timelock delay; see Table 3.

We now walk the minimal end to end with abbreviated fragments. Where a detail is needed, it is stated explicitly in the text. Code that is not essential is omitted to keep focus on integration.

Example 4 (Integrated fragments, abbreviated).

```sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

/*** 1) Governor -> Fee target (payload fixed; allowlisted; delayed) ***/
interface IFeeManager { function setFeeBps(uint16 bps) external; }
mapping(address => bool) allowedTarget;
mapping(bytes32 => uint256) eta; uint256 minDelay;

function executeSetFee(bytes32 id, address feeMgr) internal {
    // pre: forVotes >= quorum; block.timestamp >= eta[id]
    require(allowedTarget[feeMgr], "not allowed");
    bytes memory data = abi.encodeWithSelector(IFeeManager.setFeeBps.selector, 300); // 3%
    (bool ok, ) = feeMgr.call(data); require(ok, "exec failed");
}

/*** 2) Escrow with timelock as arbiter; transfers happen only in withdraw() ***/
enum S { Funded, Resolved } S s; address payer; address payee; address arbiter;
uint256 amount; mapping(address=>uint256) credit;

function resolve(bool toPayee) external {
    require(msg.sender == arbiter && s == S.Funded, "auth/state");
    s = S.Resolved; credit[toPayee ? payee : payer] += amount;
}

function withdraw() external { uint256 d=credit[msg.sender];
    require(d>0,"nothing"); credit[msg.sender]=0;
    (bool ok,)=msg.sender.call{value:d}(""); require(ok,"transfer"); }

/*** 3) Auction outcome exposed for the buyer to wire into escrow ***/
address public winner; uint256 public best; // set at Finalized (see Example 3)
```

The flow becomes straightforward.

(1) Vendor selection follows the state machine in Table 5. After the auction reaches *Finalized*, it exposes `winner` and `best`. The buyer now knows whom to pay and how much. This prepares the escrow.
(2) The buyer deploys an escrow that names `payee = winner` and sets `arbiter = timelock`. The constructor receives `best` as the deposit. The only external transfer point is `withdraw()`. This preserves the pull over push invariant from Example 1.
(3) Governance may change the platform fee. A proposal fixes the payload at creation time as ⟨target = `FeeManager`, calldata = `setFeeBps(300)`⟩. Votes are tallied using the snapshot rule from Example 2. If it passes, the governor queues the action and later calls `executeSetFee(...)` after the timelock.
(4) If a dispute occurs, anyone can propose a resolution that targets the escrow: ⟨target = `escrow`, calldata = `resolve(true)`⟩ to pay the vendor, or `resolve(false)` to refund the buyer. After voting and delay, the timelock calls `resolve(...)`. The credited party then calls `withdraw()`.

Two compact takeaways help a reader reconstruct the design.

(1) Decisions are fixed early and executed late. The governor fixes *what* will happen at proposal time and executes *after* a delay to keep effects auditable.

(2) Value moves only at one predictable point. Escrow and auction route all transfers through a single `withdraw()` function so that state transitions remain free of external calls.

REFERENCES