

BLOCKCHAIN: SMART CONTRACTS

LECTURE 5 - DESIGN PATTERNS IN SOLIDITY

FLORIN CRACIUN

IMPORTANT

**Some of the following slides are the
property of**

**Dr. Emanuel Onica & Dr. Andrei Arusoaie
Faculty of Computer Science,**

**Alexandru Ioan Cuza University of Iași
and are used with their consent.**

CONTENTS



1. Behavioral patterns
2. Upgradeability patterns
3. Economic patterns
- ~~4. Security patterns~~ (in a future lecture)

DESIGN PATTERNS IN SOLIDITY

- Plenty of less-official documentation on patterns
 - <https://github.com/fravoll/solidity-patterns>
 - <https://medium.com/@i6mi6/solidty-smart-contracts-design-patterns-ecfa3b1e9784>
 - <https://medium.com/heartbankstudio/smart-contract-design-patterns-8b7ca8b80dfb>
 - https://github.com/maxwoe/solidity_patterns
 - ...
- Papers:
 - https://eprints.cs.univie.ac.at/5665/1/bare_conf.pdf
 - <https://eprints.cs.univie.ac.at/5433/7/sanerws18iwbosemain-id1-p-380f58e-35576-preprint.pdf>
 - ...
- Various classifications of patterns



BEHAVIORAL PATTERNS

1. Guard check
 2. State machine
 3. Oracle (data provider)
 4. Randomness (data provider)
- 
- 

GUARD CHECK

- Ensure the expected behavior of a smart contract by carefully handling all the execution cases and validating input parameters
- When to use:
 - Validate user inputs
 - Check invariants
 - Rule out undesired program behaviors

GUARD CHECK EXAMPLE - MOTIVATION

```
15     function buy (uint _item_id)
16     |
17     |     public
18     |     payable
19     |     {
20     |         owner.transfer(itemsPrice[_item_id]);
21     |         // send the item to msg.sender
22     |         // ...
23     |     }
```

GUARD CHECK EXAMPLE

```
16  function buy (uint _itemId)
17      public
18      payable
19      onlyRegistered
20  {
21      require(availableItems[_itemId] >= itemsPrice[_itemId]);
22      uint senderBalance = balance[msg.sender];
23      if (senderBalance >= itemsPrice[_itemId]) {
24          owner.transfer(itemsPrice[_itemId]);
25          // send the item to msg.sender
26          // ...
27      } else {
28          revert("insufficient funds");
29      }
30      assert(balance[msg.sender] == senderBalance - itemsPrice[_itemId]);
31  }
```

modifiers

require

conditions

**revert if
conditions not met**

invariants

REQUIRE, ASSERT, REVERT

- Undo all the changes made to the state and flag an error
 - ... in the current call (and its sub-calls)
- `require`
 - Ensure valid conditions for inputs, contracts state, return values, etc.
 - Optional message: `require(condition, msg);`
- `revert`
 - Can return some meaningful information message: `revert(msg);`
 - Typically used inside an if statement
- `assert`
 - Consumes gas!
 - Recommended for internal use (e.g., testing)

STATE MACHINE PATTERN

- Contracts can go through different states; in each state certain functionality can be exposed
- When to use:
 - contract has to transition from one state to another
 - Functions should only be accessible during certain states
 - Smart contract functionality is defined by a transition system

EXAMPLE

```
3 contract Lottery {  
4   enum State {  
5     Open,  
6     Draws,  
7     Closed  
8   }
```

Multiple states: lottery open, draws, lottery closed

```
17 constructor () public {  
18   owner = msg.sender;  
19   currentState = State.Closed;  
20 }
```

Lottery state is Closed at creation time.

```
27 function openLottery()  
28   external  
29   payable  
30   onlyOwner  
31 {  
32   require(currentState == State.Closed);  
33   currentState = State.Open;  
34 }
```

**Only the owner can change the state to Open.
Also, the owner cannot open lottery of not Closed.
Now, other functions are ready for execution.**

EXAMPLE

```
56 function buyTicket(uint _ticketValue)
57     external
58     payable
59 {
60     require(msg.value == ticketPrice);
61     require(currentState == State.Open);
62     tickets[msg.sender] = _ticketValue;
63     owner.transfer(msg.value);
64 }
```

Note the requirement:
buyTicket can be
called only when
currentState is open;
otherwise it will fail!

EXAMPLE

```
36  function draws()  
37      external  
38      payable  
39      onlyOwner  
40      returns(address)  
41  {  
42      require(currentState == State.Open);  
43      currentState = State.Draws;  
44      winner = chooseWinnerRandomly();  
45      return winner;  
46  }
```

The owner can
organize draws only
in an open state!
Also, the state
changes to Draws,
and thus, it disables
buyTicket.

EXAMPLE

```
66  function closeLottery()
67      external
68      onlyOwner
69  {
70      require(currentState == State.Draws),
71      currentState = State.Closed;
72  }
```

The owner can close the lottery only after draws. The currentState changes to Closed and enables openLottery.

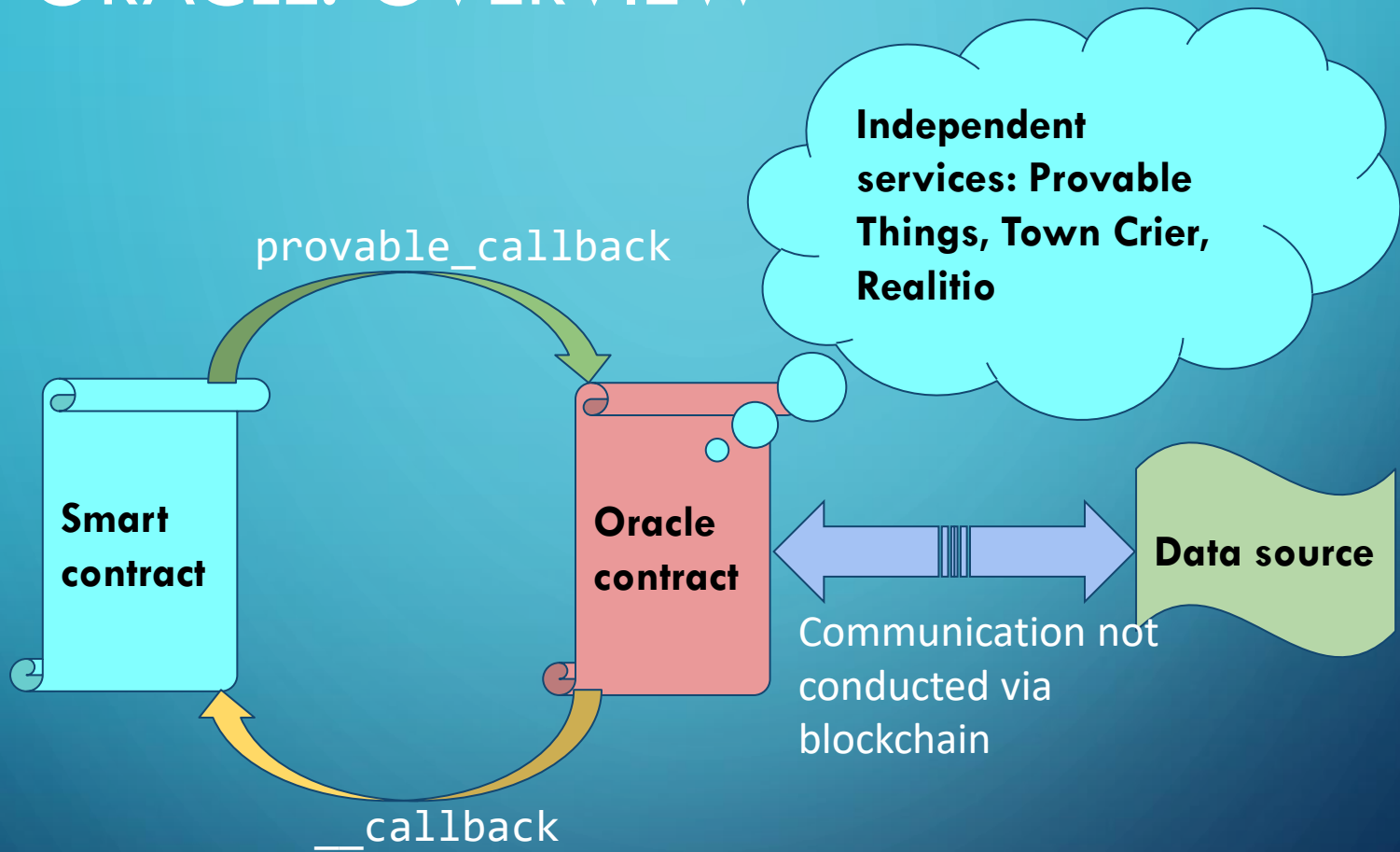
ORACLE

- Access to information outside the blockchain
- When to use
 - The smart contract relies on external information (e.g., a currency exchange rate, insurance index, etc.)
 - The external data comes from a trusted source

ORACLE

- An oracle is a 3rd party contract
 - It receives requests for external information
 - It sends the data back asynchronously
 - Typically, this is a paid service
- How to use: the contract implements 2 functions
 - A function that initiates and sends the request to the Oracle contract; this depends on the API of the Oracle contract and may send back some request id
 - A callback function that will be called by Oracle contract when it can provide the external data

ORACLE: OVERVIEW



OTHER DETAILS

- Typically the caller contract keeps a query id when calling the Oracle contract
- That id is used in the callback function to check whether the answer corresponds to that particular query id
- Also, you have to make sure that `msg.sender` is precisely the address of the Oracle contract in the callback function

RANDOMNESS

- Generate random value within a specified interval
- High demand in Ethereum
 - Games or lottery that use randomness to determine a winner
- Major challenges
 - Ethereum is deterministic, there is no randomness...
 - Determinism is important to reach consensus
 - The blockchain is public

HOW TO GENERATE RANDOM VALUES?

- Various approaches for generating randomness:
 - Use block timestamps
 - ... this is bad, miners can control those values
 - Use an oracle (we've seen that)
 - Collaborative generation of random value within the blockchain
 - Randao - this project is inactive now
 - Use the block hash as seed

(BAD) EXAMPLE

Problem 1: a miner could withhold a block if the generated number would be to his disadvantage

```
function getRandomValue()  
  public  
  view  
  returns(uint)  
{  
  return uint(blockhash(block.number - 1));  
}
```

Problem 2: block.number is a variable which can be used as input by any user. In a gambling app, an input like this always wins!

SOLUTION

- Bonneau et al. in [<https://eprint.iacr.org/2015/1008>]
 - $\text{blockNumber} = \text{current block number} + 1$
 - a trusted party T generates a seed S
 - S is sealed by hashing it together with the address of T and sent to the requesting contract
 - T waits until the next block is produced and then it reveals the seed
 - The requesting contract can validate the seed received (by hashing it with the address of T)
 - If validated the seed is hashed with the $\text{blockHash}(\text{blockNumber}) \Rightarrow$ (pseudo)random number

UPGRADEABILITY PATTERNS

- Proxy Delegate
- Eternal Storage

PROXY DELEGATE

- Upgrade your smart contract without breaking dependencies
- When to use:
 - You want to 'release' a newer version of your contract
 - You want to delegate function calls to other contracts
 - You need upgradable delegates (without breaking compatibilities)

OVERVIEW

- Participants: caller, proxy, delegate
- Steps:
 1. A caller makes a function call to the proxy contract
 2. The proxy *delegates* the call to the delegate where the actual code is located
 3. The delegate answers to the proxy, which forwards the result to the caller
- The proxy needs to distinguish among the delegate versions

PICTURES ARE SOMETIMES HELPFUL

**The Proxy Contract
should be able to switch
among the delegates.**

Caller ● — ● Proxy

Delegate 1

- Storage 1
- f() { impl 1 }

Delegate 2

- Storage 2
- f() { impl 2 }

Delegate 3

- Storage 3
- f() { impl 3 }

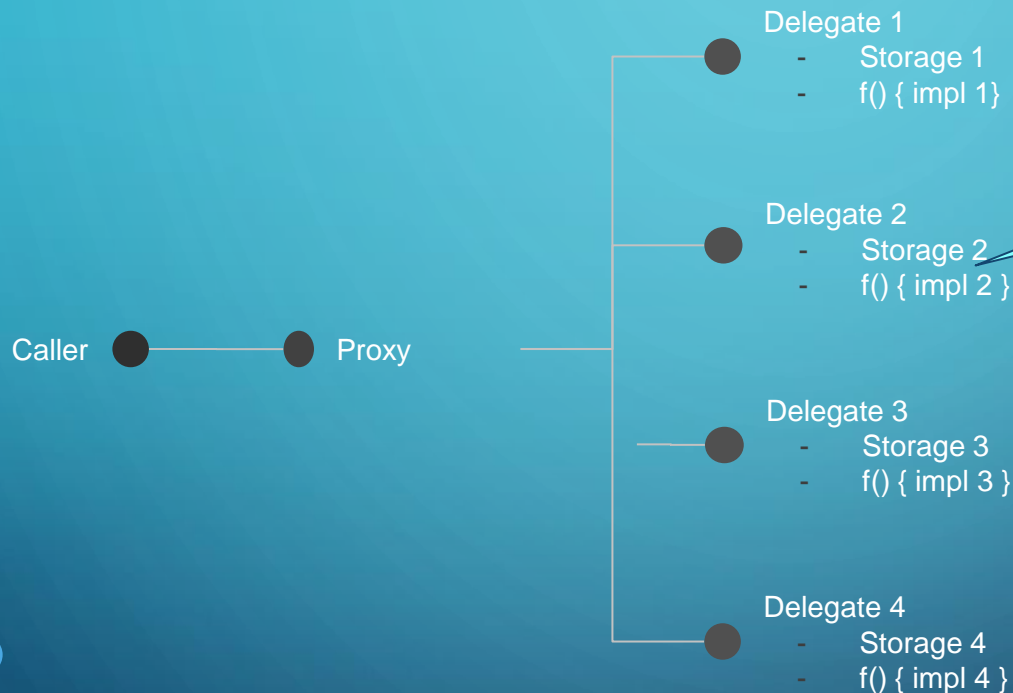
Delegate 4

- Storage 4
- f() { impl 4 }

ETERNAL STORAGE PATTERN

- Keep contract storage after smart contract upgrade
 - The older version of a contract contains storage information needed in the newer version
 - Simply copying it is costly and could produce data inconsistency
- Solution: keep storage in a separate smart contract
- When to use:
 - Your contract is upgradeable but needs the existing storage
 - You want to avoid storage migration problems

AGAIN, A PICTURE MIGHT HELP



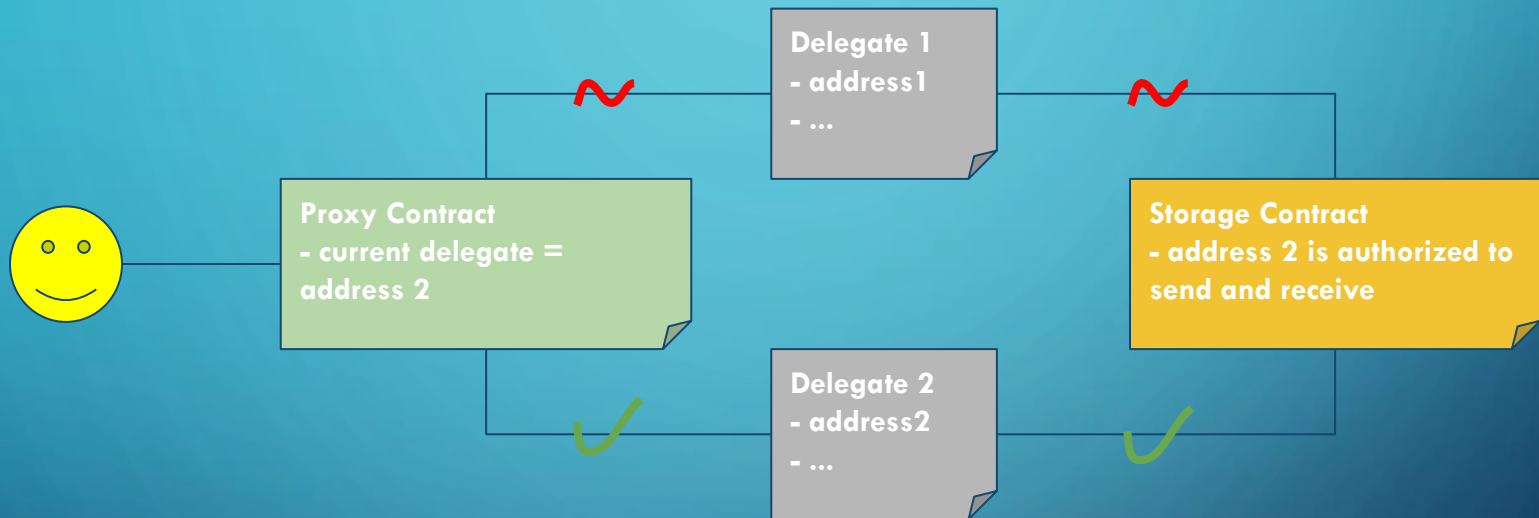
A new version still needs to keep the old storage. So, Storage 2 should include Storage 1. Solutions:

- **Copy data (costly)**
- **Use an external storage contract**

OVERVIEW

- Participants:
 - A smart contract for storage
 - An owner which does administrative work
 - Sets and upgrades the address to the latest version of a contract
 - The contract that needs the storage

COMBINING PROXY DELEGATE AND ETERNAL STORAGE



FLEXIBILITY OF THE STORAGE CONTRACT

- Smart contract for storage needs to be flexible
 - New version of smart contract may need new storage fields
 - It should be easy to add new data to it
- Solutions:
 - Implement several mappings, one for each datatype
 - Sha3 hash used as key => arbitrarily length keys
 - Functions to manage storage
 - Create
 - Retrieve
 - Update
 - Delete

ECONOMIC PATTERNS

- String Equality Comparison
- Tight Variable Packing
- Memory Array Building

Mostly concerned with optimizing gas consumption.

STRING EQUALITY COMPARISON

- Check equality is expensive in terms of gas consumption
- Are there ways to minimize the average gas consumption?
- When to use this pattern:
 - You want to check string equality
 - Most of your strings are longer than usual
 - Optimise gas consumption
- Main idea: use a slightly different algorithm for string comparison

ALGORITHM

1. First, it checks for equal length
 - Convert strings to bytes and call the builtin length member for bytes
2. Second, hash strings with keccak256()
 - If the hashes are the same, then the strings are equal \leq faster, less gas
3. Usual char by char comparison

**Are they? What about
hash collisions?**

<https://eprint.iacr.org/2011/624.pdf>

GAS COMPARISONS

Input A	Input B	Hash	Character + Length	Hash + Length
abcdefghijklmnopqrstuvwxyz	abcdefghijklmnopqrstuvwxyz	1225	7062	1261
abcdefghijklmnopqrstuvwxyzX	abcdefghijklmnopqrstuvwxyz	1225	7012	1261
Xabcdefghijklmnopqrstuvwxyz	abcdefghijklmnopqrstuvwxyz	1225	912	1261
aXabcdefghijklmnopqrstuvwxyz	abcdefghijklmnopqrstuvwxyz	1225	1156	1261
abXabcdefghijklmnopqrstuvwxyz	abcdefghijklmnopqrstuvwxyz	1225	1400	1261
abcdefghijkl	abcdefghijklmnopqrstuvwxyz	1225	690	707
a	a	1225	962	1261
ab	ab	1225	1156	1261
abc	abc	1225	1450	1261

Source: https://fravoll.github.io/solidity-patterns/string_equality_comparison.html

TIGHT VARIABLE PACKING

- Store or load statically sized-variables more efficiently w.r.t. gas consumption
- When to use:
 - You use more variables referencing structs
 - You use statically-sized arrays

OVERVIEW

- Always use the smallest data type that fits the requirements
 - E.g., uint16 instead uint256 when values are $< 2^{16}$
- Grouping data types:
 - `uint8 a; uint8 b; uint8 c; uint8 d;`
 - fill only one 32 byte slot in EVM
- Problems:
 - It needs to design your smart contract storage with variable packing in mind

MEMORY ARRAY BUILDING

- Aggregate and retrieve data from contract efficiently w.r.t. gas consumption
- When to use:
 - Get aggregated data from storage
 - Avoid paying gas when reading data

TIPS

- `public` => free access to the value of the variable
- `view` => aggregate and read data from contract storage without associated cost;
 - Lookup in array => an array is rebuild in the memory instead of storage
 - Everything is done on the local node; no requests to other nodes to get blockchain data

OTHER PATTERNS

- Mortal contracts => selfdestruct
- Lifetime
 - automatic expiration of a contract after a specified amount of time
- Maintenance -> upgradeability patterns
- Ownership