LimeChain

Telerik Academy

Smart contracts design patterns

Lime Academy - Season 1

Security patterns

- Access restriction
- Withdraw (ETH transfers)
- Checks-effects-interactions
- Emergency stop (Pausable)

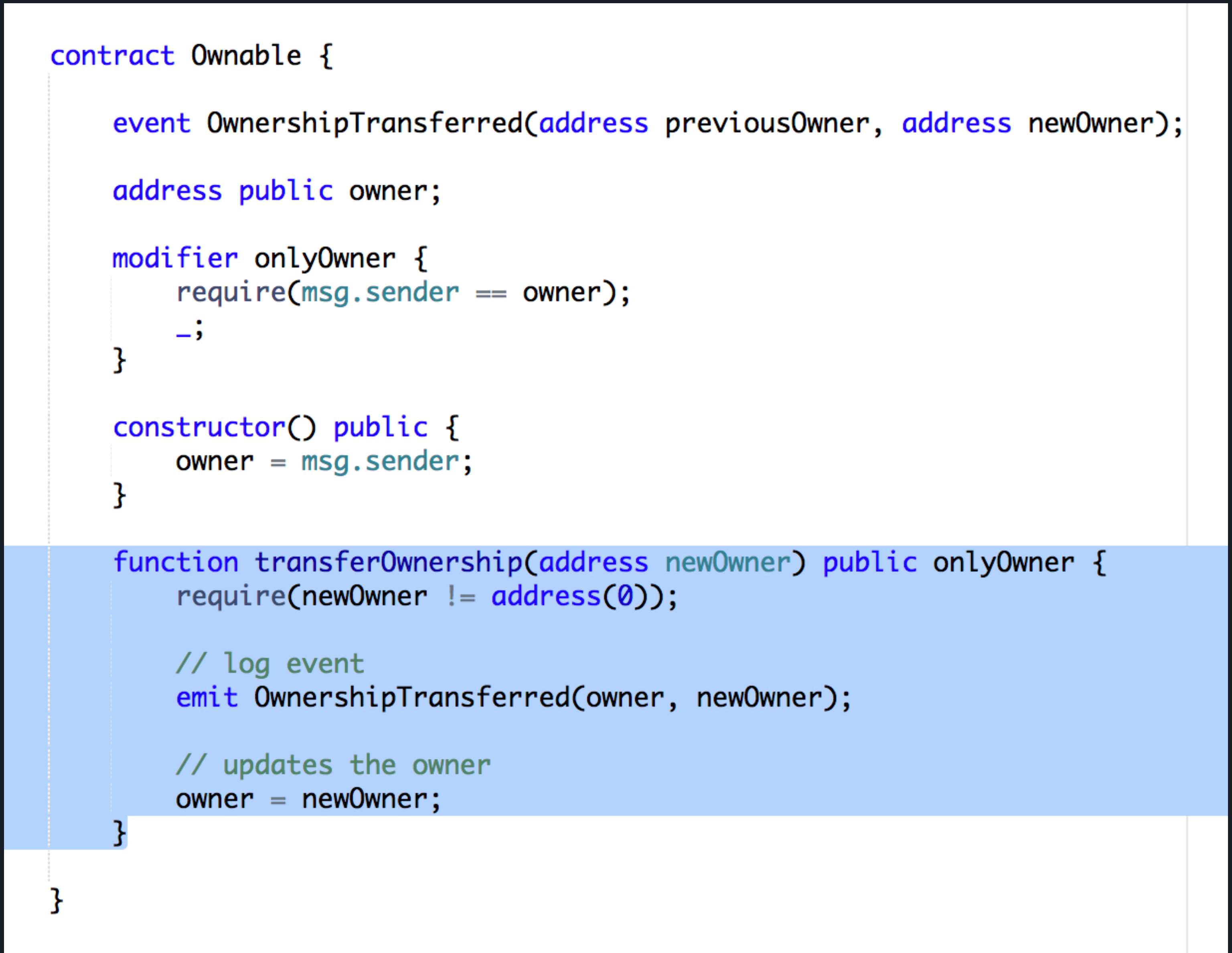
Access restriction

Intent

Restrict the access to contract functionality according to suitable criteria (increasing security against unauthorized access)

Implementation

- function modifiers
- require() statements



```
contract Ownable {

    event OwnershipTransferred(address previousOwner, address newOwner);

    address public owner;

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    constructor() public {
        owner = msg.sender;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0));

        // log event
        emit OwnershipTransferred(owner, newOwner);

        // updates the owner
        owner = newOwner;
    }

}
```

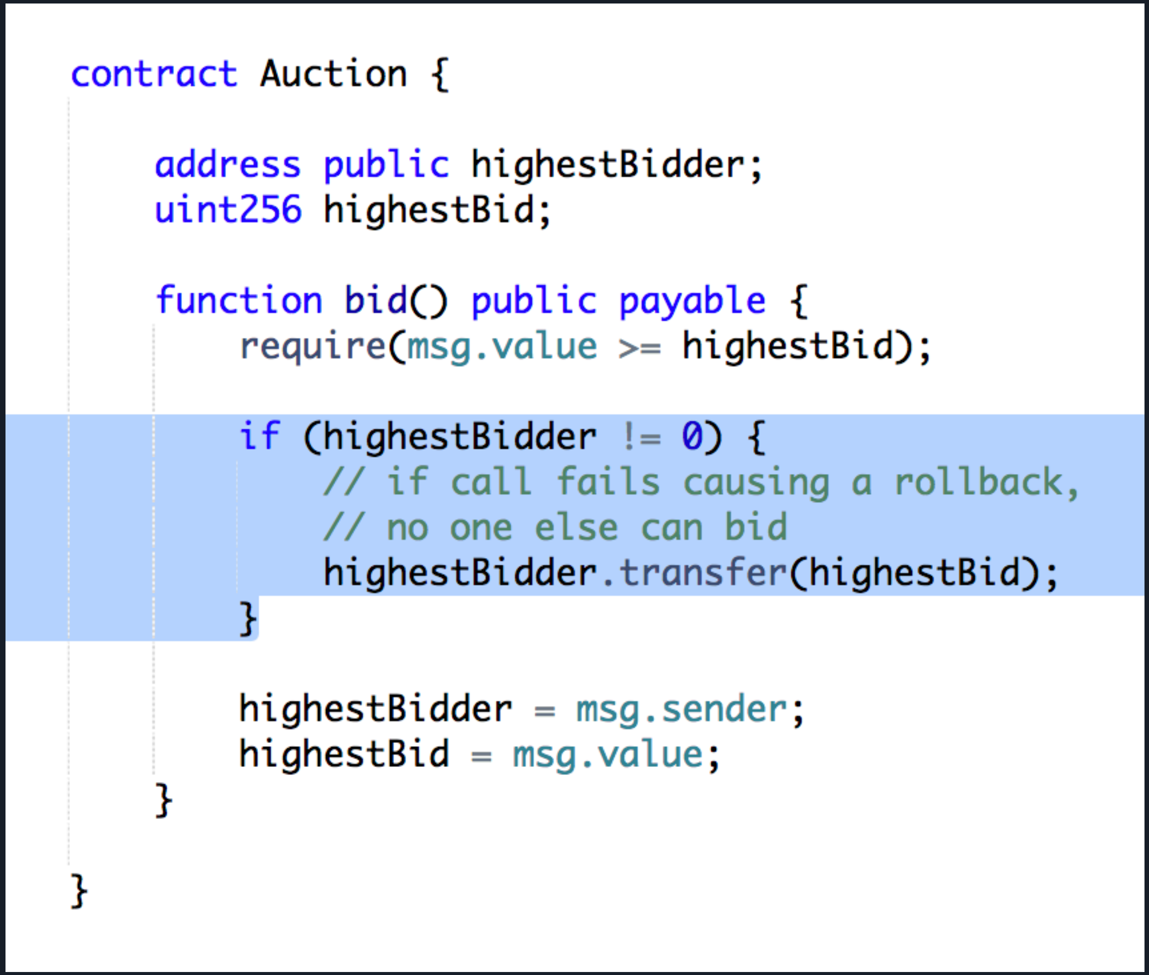
Withdraw (ETH transfers)

Intent

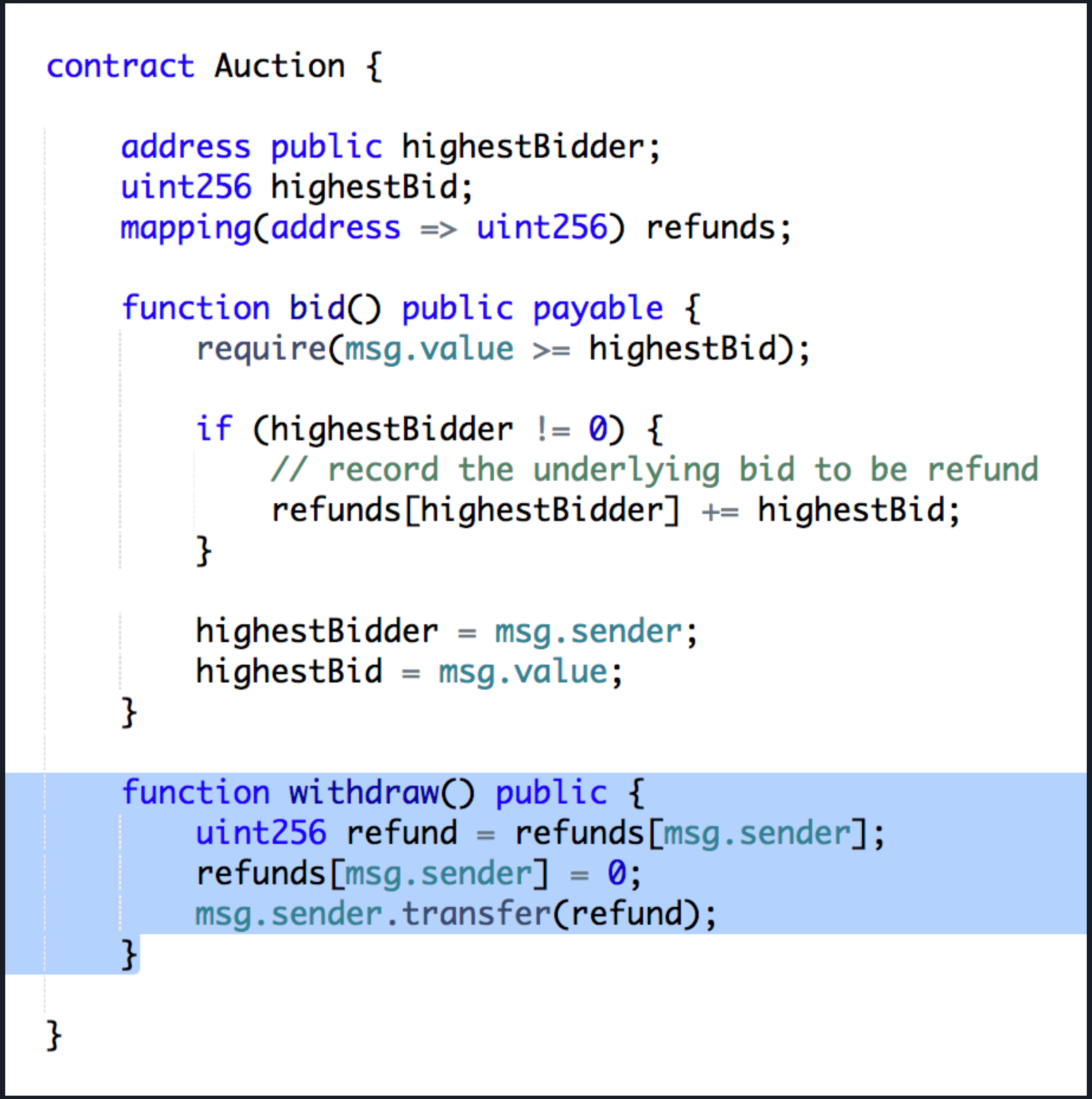
- Shift the risk associated with transferring ether to the user
- Avoid handling of multiple ether transfers within one function call (possible deadlocks)

Implementation

Isolating the external call into its own function / transaction that can be initiated by the recipient of the call



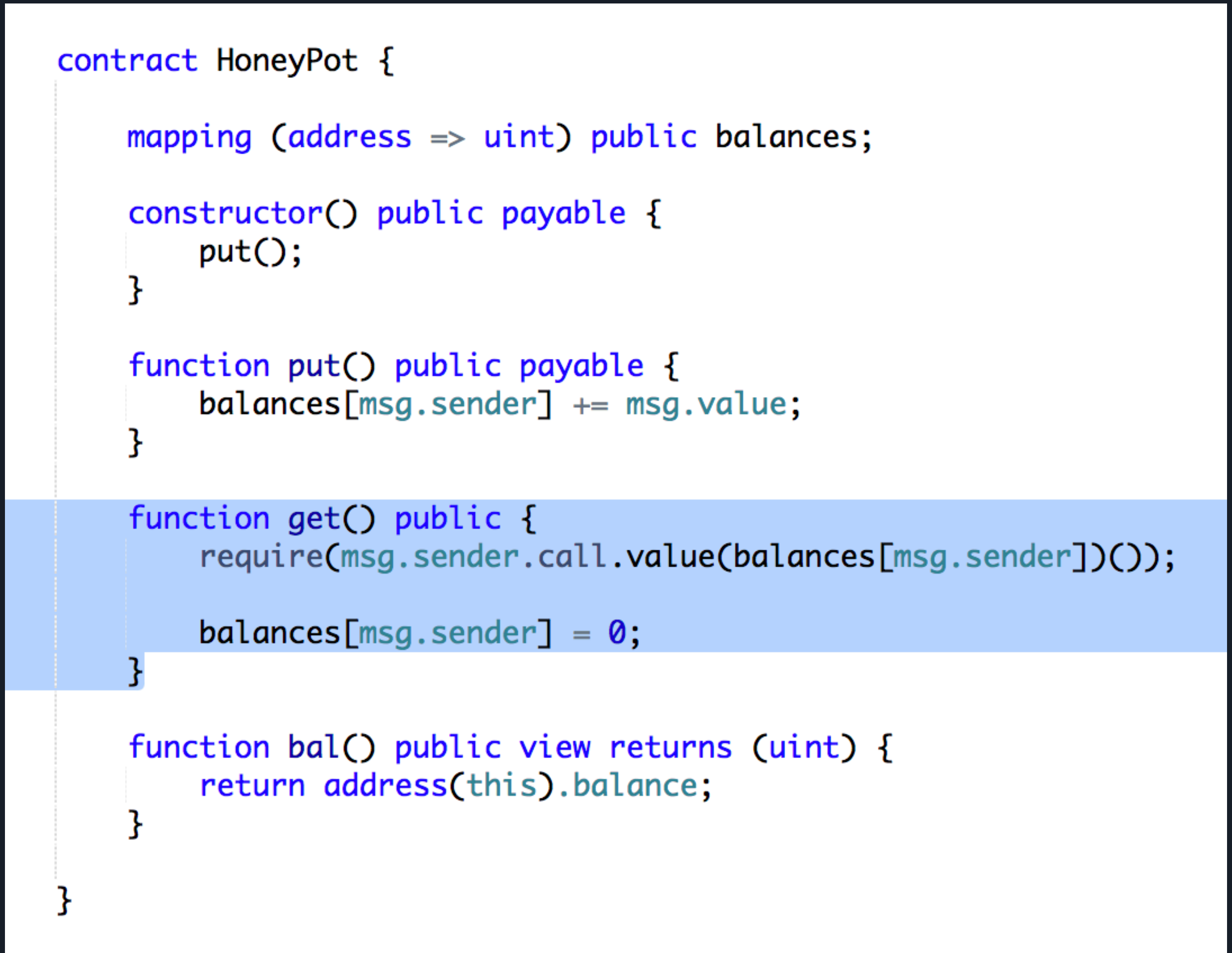
```
contract Auction {

    address public highestBidder;
    uint256 highestBid;

    function bid() public payable {
        require(msg.value >= highestBid);

        if (highestBidder != 0) {
            // if call fails causing a rollback,
            // no one else can bid
            highestBidder.transfer(highestBid);
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }

}
```



```
contract Auction {

    address public highestBidder;
    uint256 highestBid;
    mapping(address => uint256) refunds;

    function bid() public payable {
        require(msg.value >= highestBid);

        if (highestBidder != 0) {
            // record the underlying bid to be refund
            refunds[highestBidder] += highestBid;
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }

    function withdraw() public {
        uint256 refund = refunds[msg.sender];
        refunds[msg.sender] = 0;
        msg.sender.transfer(refund);
    }

}
```

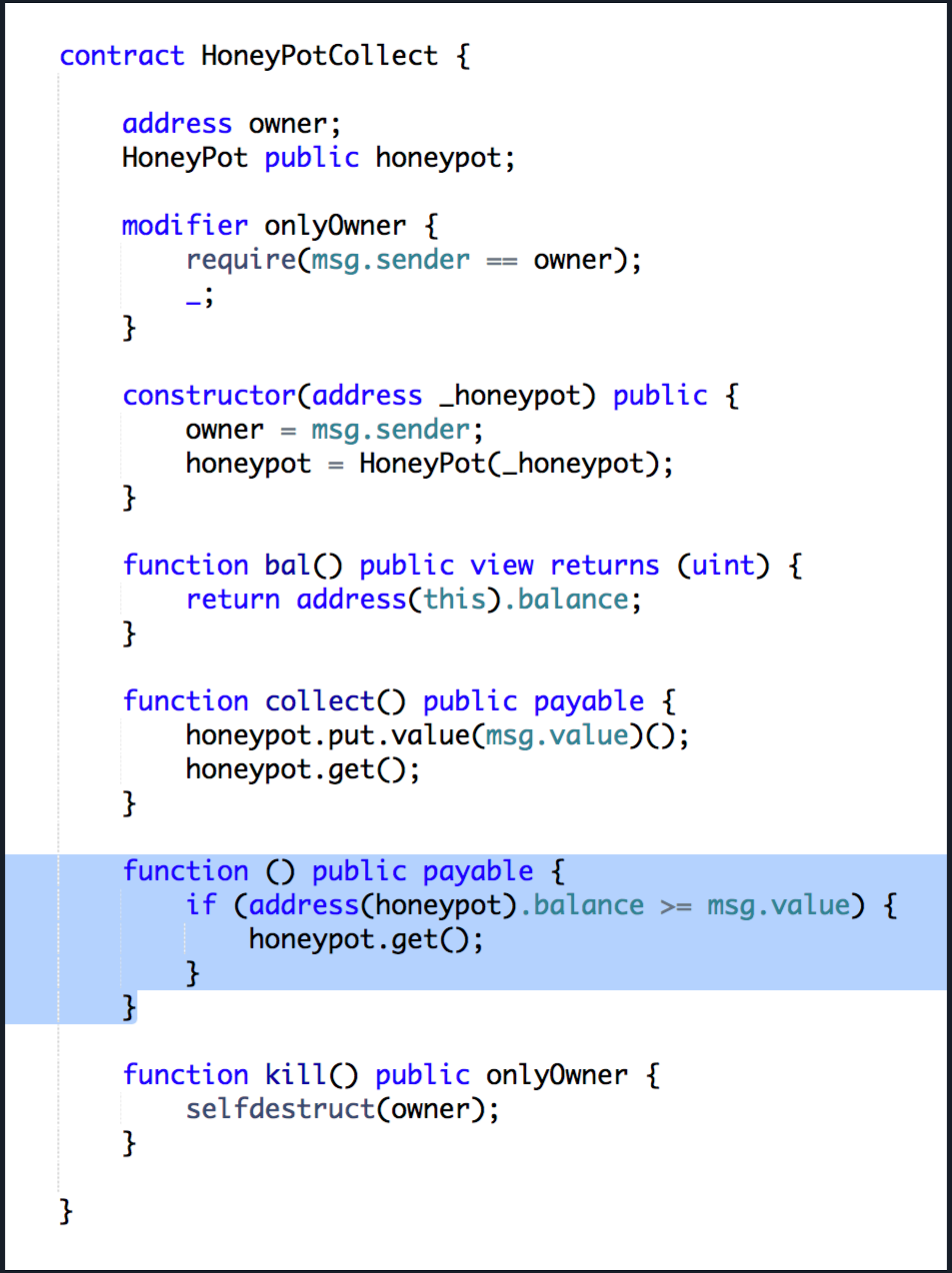
Reentrancy

Vulnerable contract



```
contract HoneyPot {

    mapping (address => uint) public balances;

    constructor() public payable {
        put();
    }

    function put() public payable {
        balances[msg.sender] += msg.value;
    }

    function get() public {
        require(msg.sender.call.value(balances[msg.sender])());

        balances[msg.sender] = 0;
    }

    function bal() public view returns (uint) {
        return address(this).balance;
    }

}
```

Malicious contract



```
contract HoneyPotCollect {

    address owner;
    HoneyPot public honeypot;

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    constructor(address _honeypot) public {
        owner = msg.sender;
        honeypot = HoneyPot(_honeypot);
    }

    function bal() public view returns (uint) {
        return address(this).balance;
    }

    function collect() public payable {
        honeypot.put.value(msg.value)();
        honeypot.get();
    }

    function () public payable {
        if (address(honeypot).balance >= msg.value) {
            honeypot.get();
        }
    }

    function kill() public onlyOwner {
        selfdestruct(owner);
    }

}
```

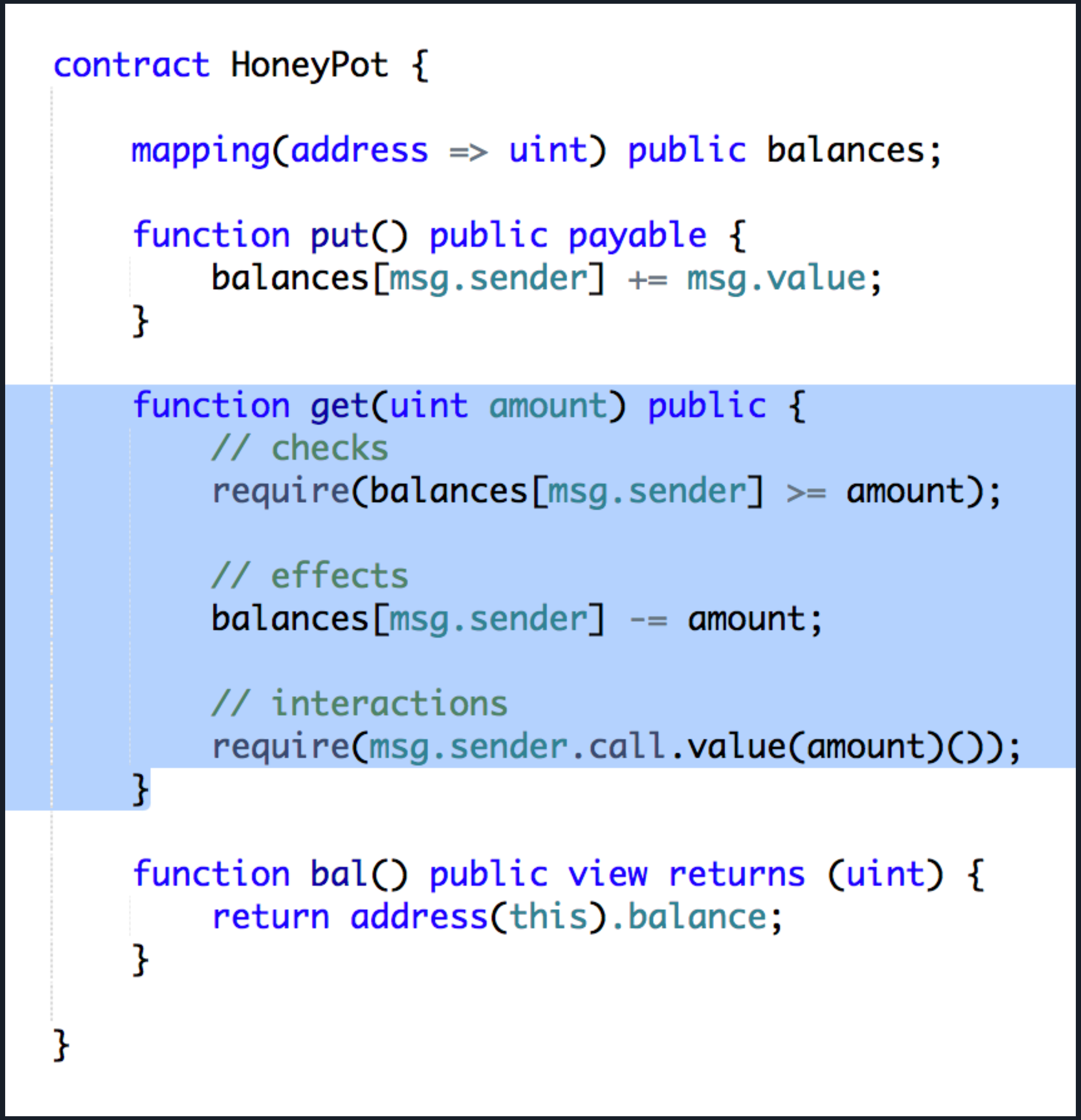
Checks-effects-interactions

Intent

Reduce the attack surface for malicious contract trying to hijack control flow after an external call (re-entrancy attacks)

Implementation

- Checks - execute checks whether this function can be called (better use modifiers)
- Effects - update internal contract state
- Interactions - execute external calls/transfers



```
contract HoneyPot {

    mapping(address => uint) public balances;

    function put() public payable {
        balances[msg.sender] += msg.value;
    }

    function get(uint amount) public {
        // checks
        require(balances[msg.sender] >= amount);

        // effects
        balances[msg.sender] -= amount;

        // interactions
        require(msg.sender.call.value(amount)());
    }

    function bal() public view returns (uint) {
        return address(this).balance;
    }

}
```

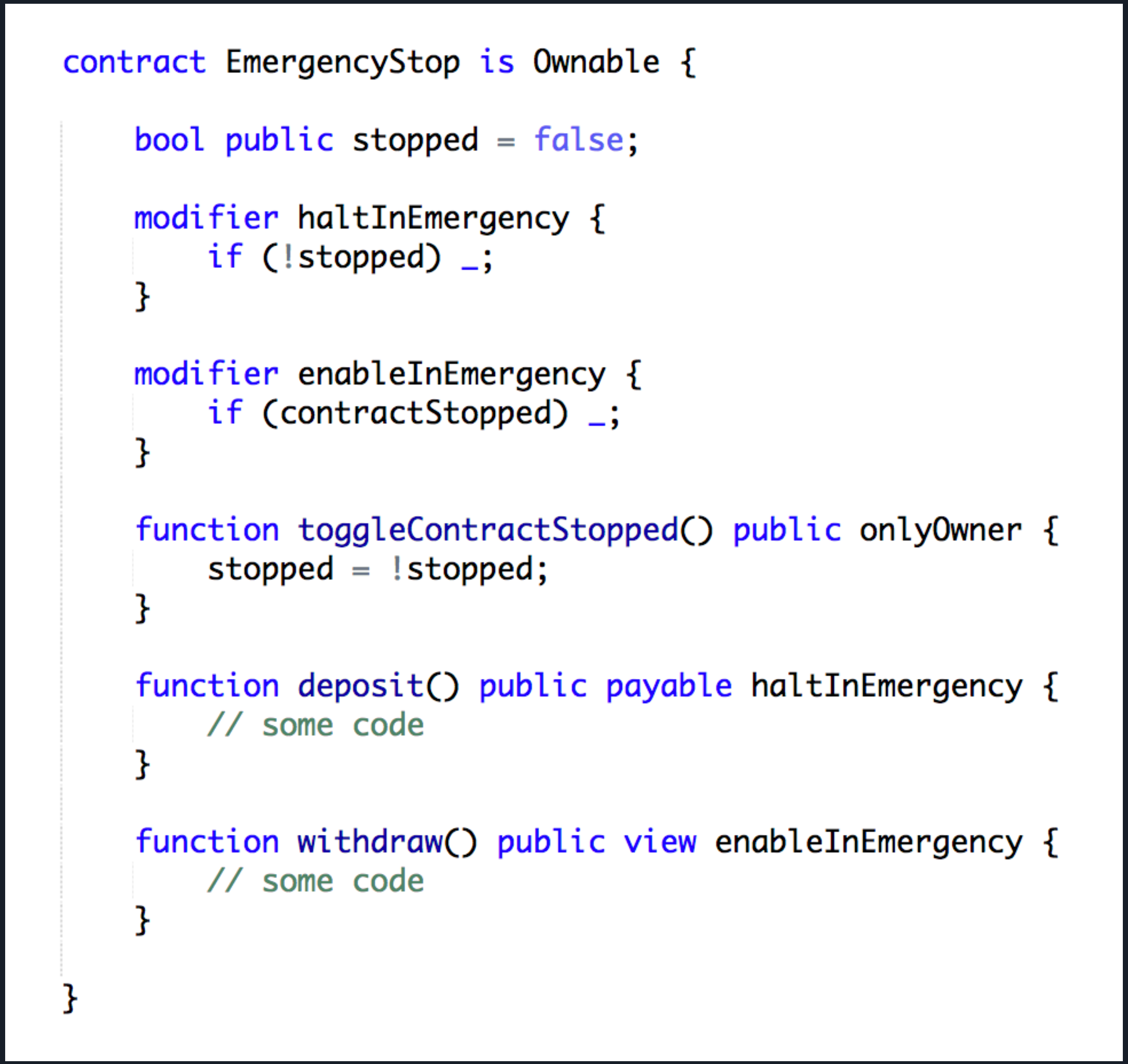
Emergency stop (Pausable)

Intent

Disable critical contract functionality in case of an emergency (halt its execution in case of a major bug or security issue)

Implementation

Mechanism allowing contract owner to switch from/to disabled contract state



```
contract EmergencyStop is Ownable {

    bool public stopped = false;

    modifier haltInEmergency {
        if (!stopped) _;
    }

    modifier enableInEmergency {
        if (contractStopped) _;
    }

    function toggleContractStopped() public onlyOwner {
        stopped = !stopped;
    }

    function deposit() public payable haltInEmergency {
        // some code
    }

    function withdraw() public view enableInEmergency {
        // some code
    }

}
```

Behavioral patterns

- Guard check (input/state validation)
- Factory / Registry
- Oracles
- Proxy / Delegate

Guard check

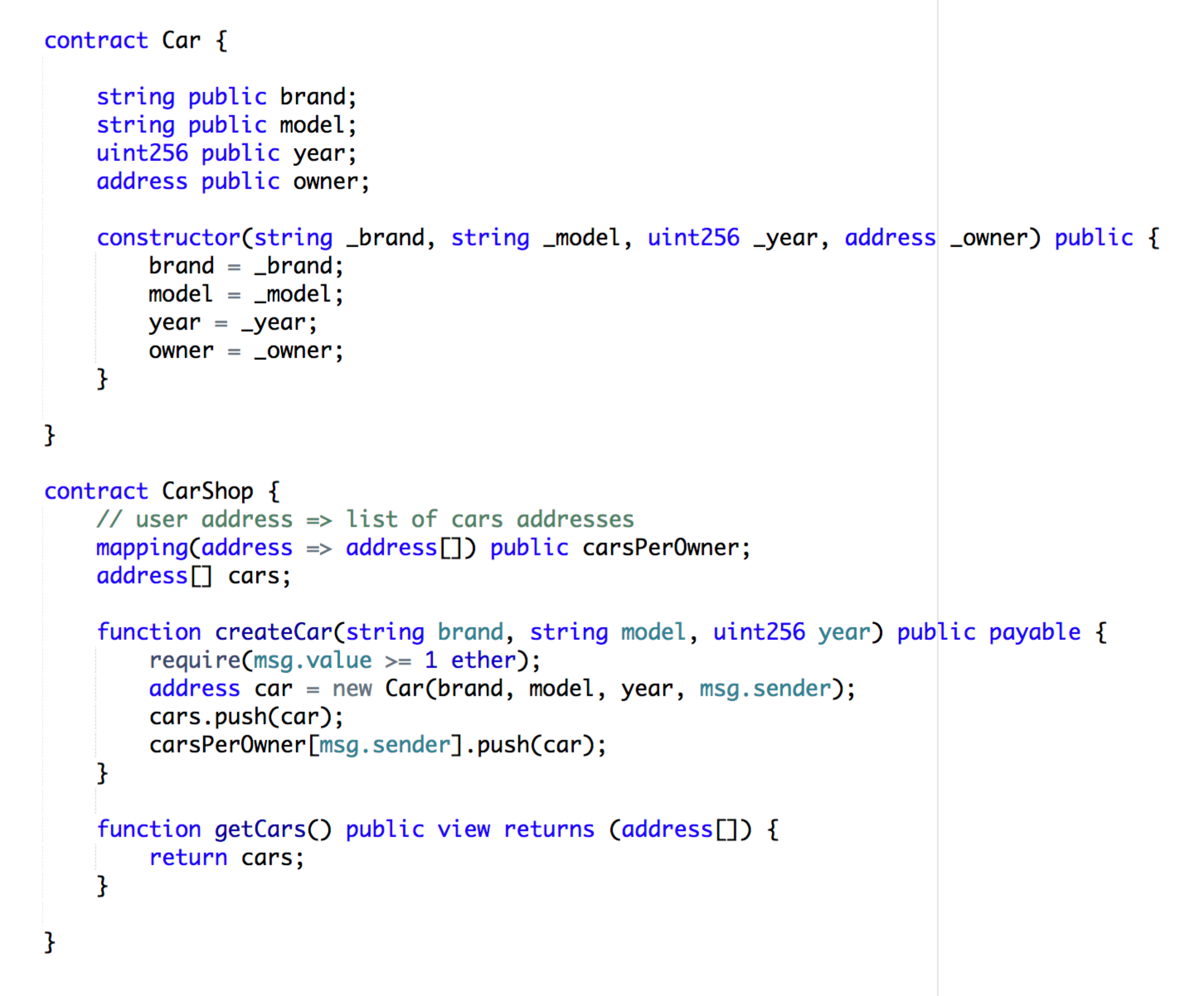
Intent

Ensure that the behavior of a smart contract and its input parameters are as expected.

Implementation

- function modifiers
- require()
- assert()

Factory / Registry



```
contract Car {

    string public brand;
    string public model;
    uint256 public year;
    address public owner;

    constructor(string _brand, string _model, uint256 _year, address _owner) public {
        brand = _brand;
        model = _model;
        year = _year;
        owner = _owner;
    }

}

contract CarShop {
    // user address => list of cars addresses
    mapping(address => address[]) public carsPerOwner;
    address[] cars;

    function createCar(string brand, string model, uint256 year) public payable {
        require(msg.value >= 1 ether);
        address car = new Car(brand, model, year, msg.sender);
        cars.push(car);
        carsPerOwner[msg.sender].push(car);
    }

    function getCars() public view returns (address[]) {
        return cars;
    }

}
```

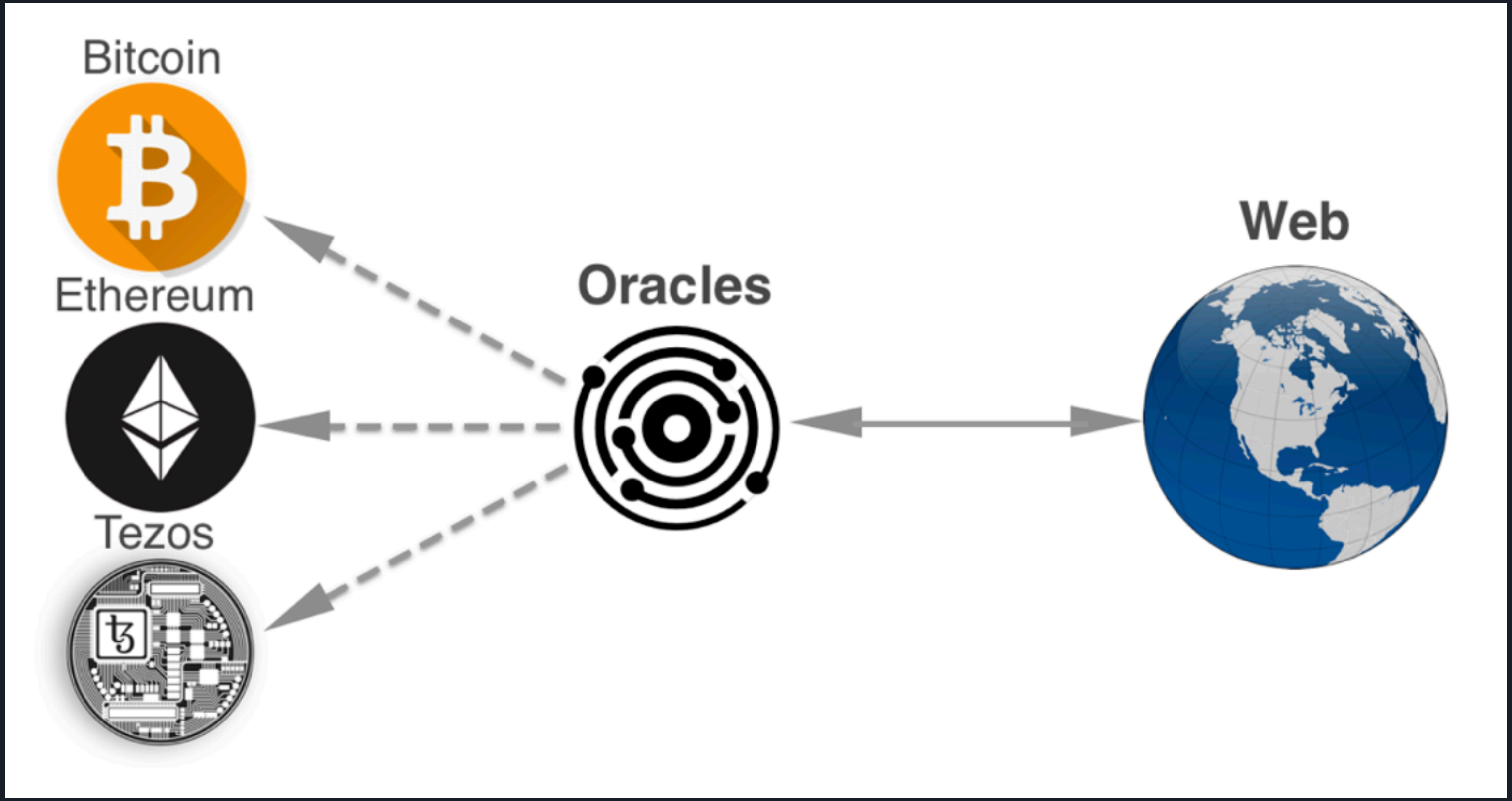
Oracles

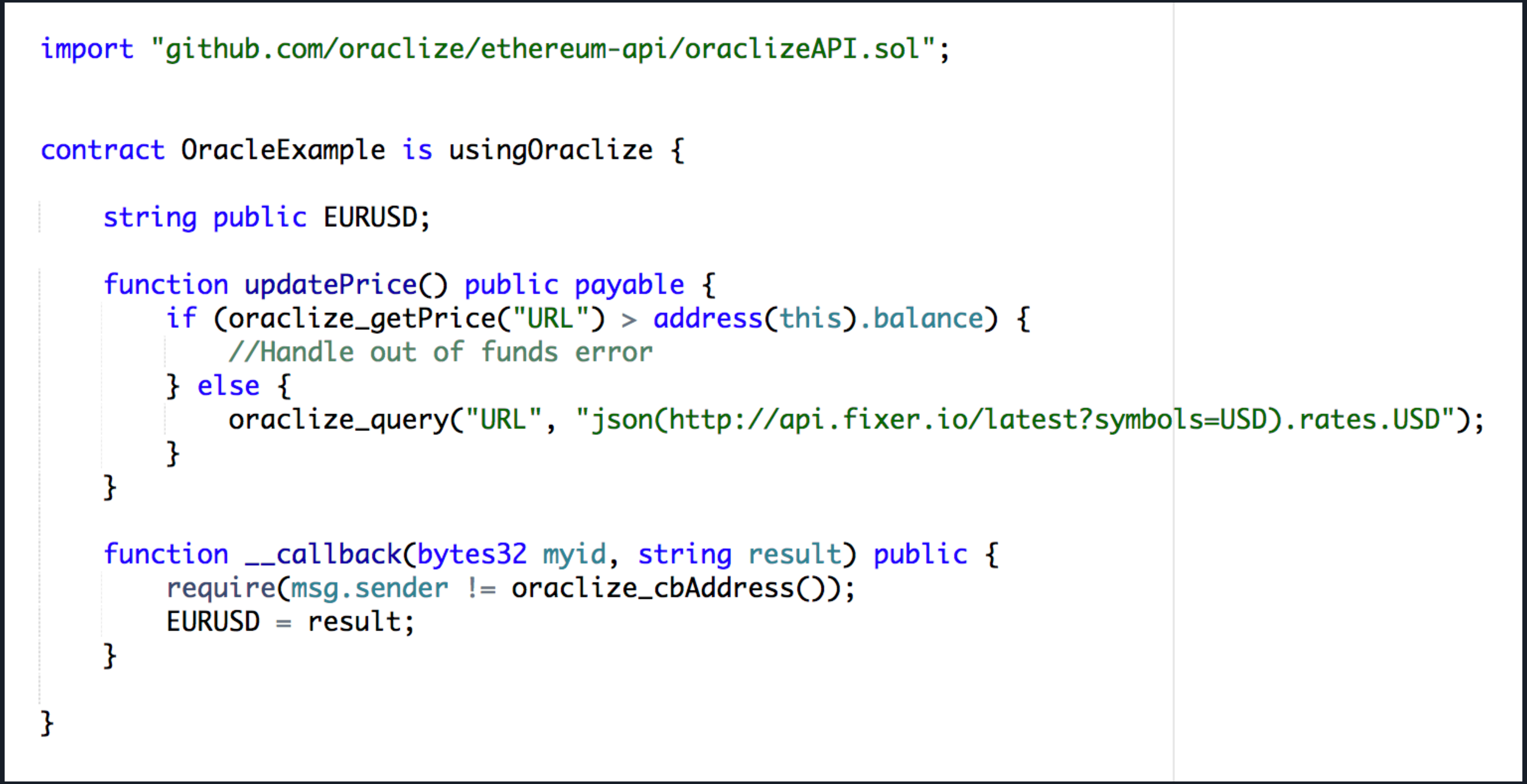
Intent

Gain access to data stored outside of the blockchain

Implementation

- Using oraclizeAPI (API to agent living on the blockchain and providing information in the form of responses to queries)
- Logic (outside blockchain) recurrently calling & updating contract state


Bitcoin
Ethereum
Tezos
Oracles
Web



```
import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";


contract OracleExample is usingOraclize {

    string public EURUSD;

    function updatePrice() public payable {
        if (oraclize_getPrice("URL") > address(this).balance) {
            //Handle out of funds error
        } else {
            oraclize_query("URL", "json(http://api.fixer.io/latest?symbols=USD).rates.USD");
        }
    }

    function __callback(bytes32 myid, string result) public {
        require(msg.sender != oraclize_cbAddress());
        EURUSD = result;
    }

}
```

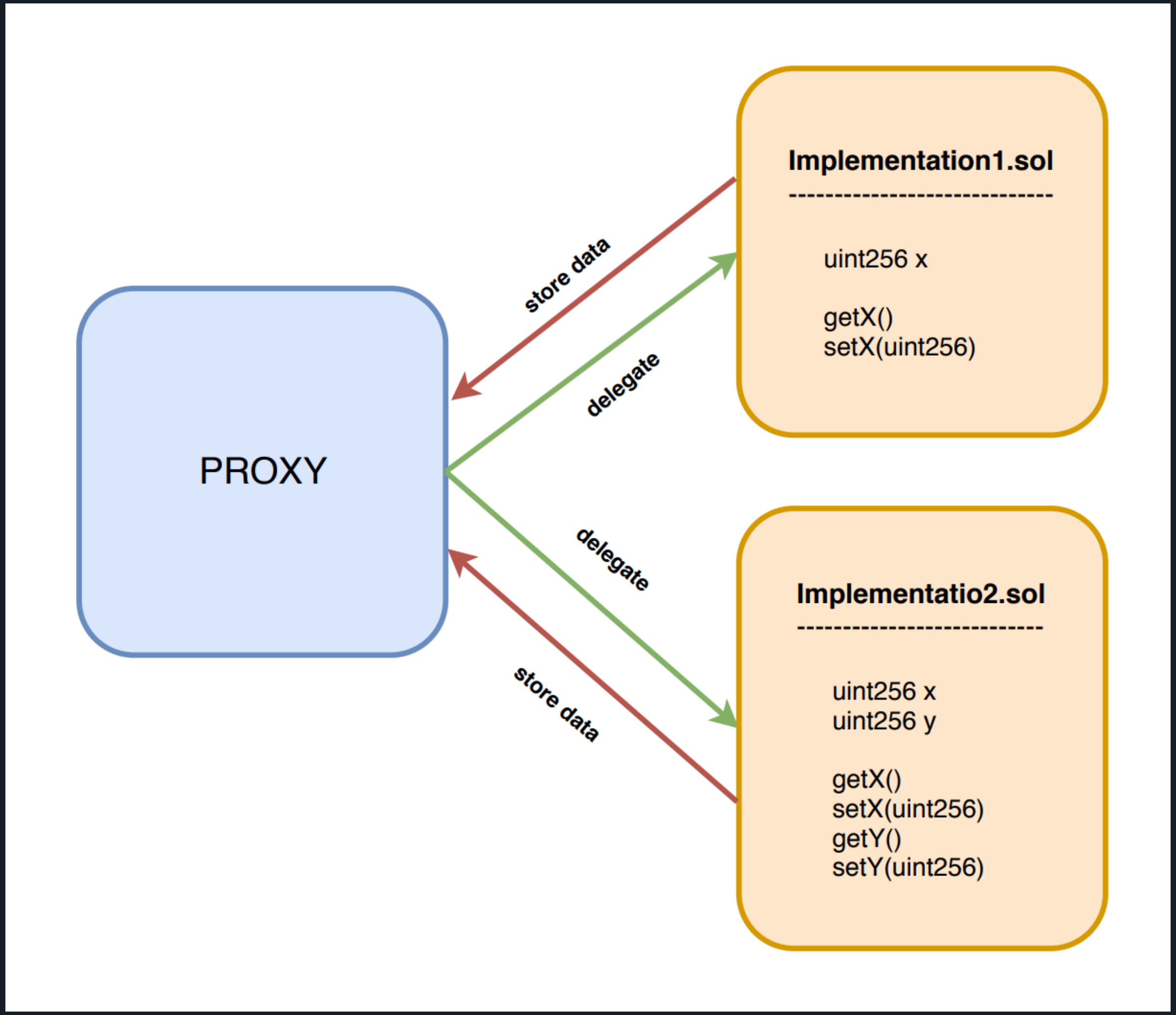
Proxy / Delegate

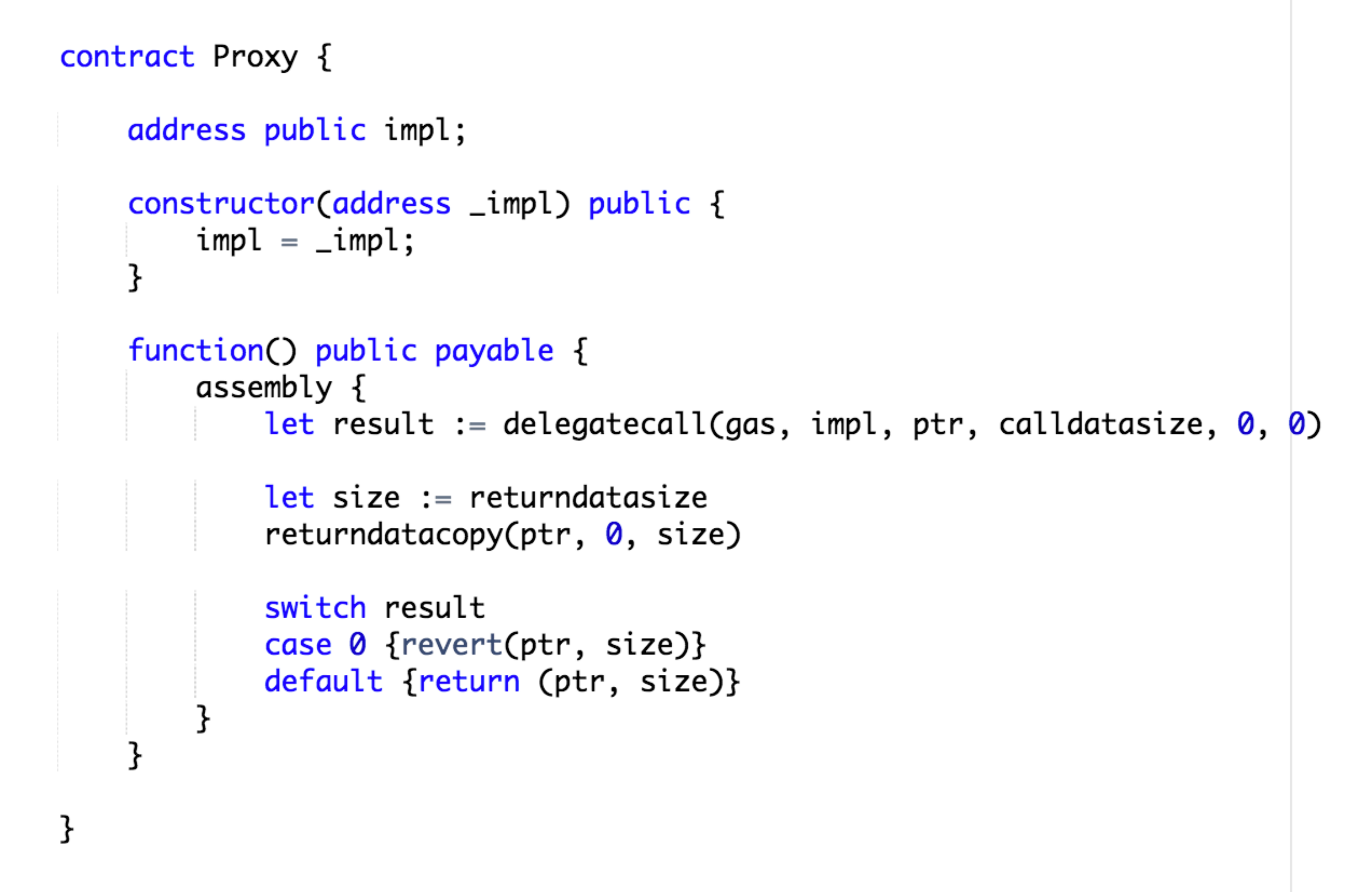
Intent

Allow to upgrade smart contracts without breaking any dependencies and loose any data

Implementation

- “Redirect” or “delegate” calls to the contract which should execute the logic and also
- Store the result of execution in the proxy storage so that we won’t lose data when upgrading to the new implementation contract


PROXY
store data
delegate
delegate
store data
Implementation1.sol
----
uint256 x
getX()
setX(uint256)
Implementatio2.sol
----
uint256 x
uint256 y
getX()
setX(uint256)
getY()
setY(uint256)

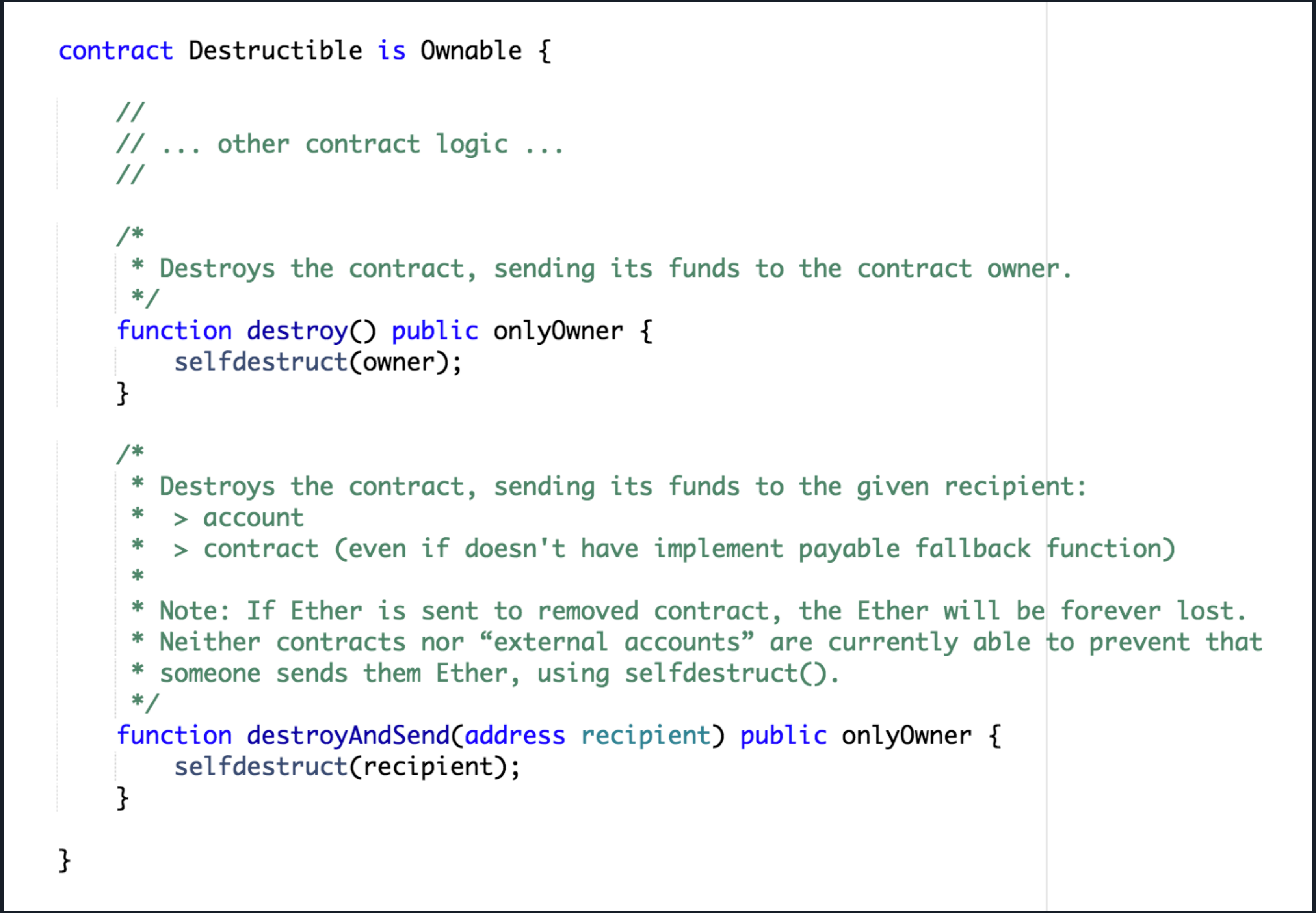


```
contract Proxy {

    address public impl;

    constructor(address _impl) public {
        impl = _impl;
    }

    function() public payable {
        assembly {
            let result := delegatecall(gas, impl, ptr, calldatasize, 0, 0)

            let size := returndatasize
            returndatacopy(ptr, 0, size)

            switch result
            case 0 {revert(ptr, size)}
            default {return (ptr, size)}
        }
    }

}
```

Lifecycle patterns

- Mortable (Destructible)

Mortal (Destructible)



```
contract Destructible is Ownable {

    //
    // ... other contract logic ...
    //

    /*
     * Destroys the contract, sending its funds to the contract owner.
     */
    function destroy() public onlyOwner {
        selfdestruct(owner);
    }

    /*
     * Destroys the contract, sending its funds to the given recipient:
     *   > account
     *   > contract (even if doesn't have implement payable fallback function)
     *
     * Note: If Ether is sent to removed contract, the Ether will be forever lost.
     * Neither contracts nor “external accounts” are currently able to prevent that
     * someone sends them Ether, using selfdestruct().
     */
    function destroyAndSend(address recipient) public onlyOwner {
        selfdestruct(recipient);
    }

}
```

Homework

- Try to apply already discussed patterns to CryptoCars project (only applicable)

Further reading

- Design Patterns for Smart Contracts in the Ethereum Ecosystem
  https://eprints.cs.univie.ac.at/5665/1/bare_conf.pdf
- Security Patterns in the Ethereum Ecosystem and Solidity
  https://eprints.cs.univie.ac.at/5433/7/sanerws18iwbosemain-id1-p-380f58e-35576-preprint.pdf
- Solidity patterns
  https://fravoll.github.io/solidity-patterns/
- Solidity by example
  https://github.com/raineorshine/solidity-by-example

Q & A

Telerik Academy

Thanks !

vlado@limechain.tech