Ethereum Smart Contracts

Gallersdörfer, U., Holl, P., & Matthes, F. (2019). "Blockchain-based Systems Engineering“. Lecture Slides. TU Munich.

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

Outline

1. Solidity Introduction
   - Definition
   - Anatomy of a smart contract
   - Language features
   - Functions
   - Modifiers
   - Inheritance
   - Abstract contracts and interfaces

2. Designing Smart Contracts
   - Problem assessment
   - Modelling entities
   - Modelling transactions

3. Cross-contract and blockchain interaction
   - EVM contract function execution
   - Transactions and messages
   - Address class
   - Message object
   - Block object
   - Transaction object

Definition

- Solidity is a **high-level language** to write **smart contracts** for **Ethereum.**
- **Contracts** can be defined as **encapsulated units**, similar to classes in traditional **object-oriented** programming languages like Java.
- A **contract** has **its own, persistent state** on the blockchain which is **defined by state variables** in the contract.
- **Functions** are **used to change** the **state** of the smart contract or to **perform** other **computations.**
- Solidity is **compiled to bytecode** which is **persistent and immutable** once deployed to the blockchain:
  ➔ **No patch deployment** possible
  ➔ **Smart contracts** must be **perfect before** using them in **production**!

From Solidity source code to a deployed smart contract

- **Solidity code** is stored in files with the special **file extension .sol.**
- A **good practice** is to have one **separate .sol file per contract.**
- The **Solidity compiler** takes a .sol file as input and **generates** the corresponding sequence of **EVM opcode instructions.**
- The **opcode instructions** are then encoded **as hex bytecode.**
- The **contract** is **deployed** via a **special transaction** containing the bytecode as payload.
- **Once** the transaction is **mined**, a **new contract account** on the Ethereum network **is created.**
- The contract is now usable.

Anatomy of a Solidity smart contract file

File: **BBSE.sol**

```
contract BBSE {

    struct Tutor {
        string firstName;
        string lastName;
    }
    mapping (address => Tutor) tutors;
    address professor;

    modifier onlyProfessor {
        require(msg.sender == professor);
        _;
    }

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

    function getProfessor() pure returns (address) {
        return professor;
    }

    // This function adds a new tutor
    function addTutor(address tutorAddress,
    string firstName, string lastName) onlyProfessor {
        var tutor = tutors[tutorAddress];
        tutor.firstName = firstName;
        tutor.lastName = lastName;
    }

}
```

State variables

- State variables are permanently stored in the contract's storage.
- Changing the state requires a transactions and therefore costs ether.
- Reading the state of a contract is free and does not require a transaction.

Anatomy of a Solidity smart contract file

File: **BBSE.sol**

```
contract BBSE {

    struct Tutor {
        string firstName;
        string lastName;
    }
    mapping (address => Tutor) tutors;
    address professor;

    modifier onlyProfessor {
        require(msg.sender == professor);
        _;
    }

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

    function getProfessor() pure returns (address) {
        return professor;
    }

    // This function adds a new tutor
    function addTutor(address tutorAddress,
    string firstName, string lastName) onlyProfessor {
        var tutor = tutors[tutorAddress];
        tutor.firstName = firstName;
        tutor.lastName = lastName;
    }

}
```

Function modifiers

- Function modifiers are a convenient way to reuse pieces of code.
- Changes the behavior of a function.
- Can execute code either before and/or after the actual function execution.
- The low dash _ indicates where the actual function code is injected.
- Often used for authentication.

Anatomy of a Solidity smart contract file

File: **BBSE.sol**

```
contract BBSE {

    struct Tutor {
        string firstName;
        string lastName;
    }
    mapping (address => Tutor) tutors;
    address professor;

    modifier onlyProfessor {
        require(msg.sender == professor);
        _;
    }

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

    function getProfessor() pure returns (address) {
        return professor;
    }

    // This function adds a new tutor
    function addTutor(address tutorAddress,
    string firstName, string lastName) onlyProfessor {
        var tutor = tutors[tutorAddress];
        tutor.firstName = firstName;
        tutor.lastName = lastName;
    }

}
```

Constructor

- The constructor function is executed once when the contract is created through a transaction.
- The function cannot be called after the creation of the contract.
- Usually used to initialize the state of a smart contract.
- Execution costs gas and more complex constructors lead to higher deployment costs.

Anatomy of a Solidity smart contract file

File: **BBSE.sol**

```
contract BBSE {

    struct Tutor {
        string firstName;
        string lastName;
    }
    mapping (address => Tutor) tutors;
    address professor;

    modifier onlyProfessor {
        require(msg.sender == professor);
        _;
    }

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

    function getProfessor() pure returns (address) {
        return professor;
    }

    // This function adds a new tutor
    function addTutor(address tutorAddress,
    string firstName, string lastName) public onlyProfessor {
        var tutor = tutors[tutorAddress];
        tutor.firstName = firstName;
        tutor.lastName = lastName;
    }

}
```

Functions

- Functions are used to change the state of a contract.
- Can also be used to read the state of the contract.
- Consist of a name, a signature, a visibility, a type, a list of modifiers, and a return type.

Formal definition:

function (<parameter types>)
{internal|external|public|private}
[pure|constant|view|payable]
[(modifiers)]
[returns (<return types>)]

Language feature overview

Solidity is **inspired by JavaScript** and comes with a very similar syntax. Furthermore, it implements the standard set of features for high-level (object-oriented) programming languages. Compared to the dynamically-typed JavaScript, Solidity uses static types.

Built-in data types

int, uint, bool, array, struct, enum, mapping

Built-in first level objects

block, msg, tx, address

Built-in functions

Error handling: assert(), require(), revert()
Math & Crypto: addmod(), mulmod(), sha3(), keccak256(), sha256(), ripemd160(), ecrecover()
Information: gasleft(), blockhash()
Contract related: selfdestruct()

A set of literals

Solidity comes with some Ethereum specific literals (like eth for units, e.g., int a = 5 eth)

Flow control

if, else, do, while, break, continue, for, return, ? … : … (ternary operator)

Function and variable visibility

In Solidity, functions can be declared with four different visibility types.

External
External methods can be called by other contracts and via transactions issued by a certain wallet. Methods declared as external **are always publicly visible** and **can’t be called directly by the contract itself**.

Public
Public can **be called internally** by the contract itself but also **externally** by other contracts and via transactions. **State variable** which are defined as public will **by default have getter and setter** methods created automatically by the compiler.

Internal
Internal methods can only be accessed by the contract itself or by any contract derived from it. They are not callable from other contracts nor via transactions.

Private
Private methods can **only** be called **internally** by the contract who owns the method. **Derived contracts cannot access** a private method of their parent contract.

Special function types

Solidity provides two special function type declaration besides the default one.

View function

Functions which are declared as view are read only, i.e. they do not modify any state variable nor alter the state of the blockchain. However, they can read from state variables.

```
uint state = 5;
function add(uint a, uint b) public view returns (uint sum) { return a + b + state }
```

Pure function

Pure functions can be seen as a subset of view functions which don't modify the state but also don't read from the state.

```
function add(uint a, uint b) public pure returns (uint sum) { return a +b }
```

Fallback function

A contract can have one unnamed fallback function. The fallback function is called when no other function matches the function call, e.g. when Ether is sent without any function call. A special feature of this function is that it can't have any parameters and doesn't return anything.

```
function() { /* ... */ }
```

Special function types (cont.)

Payable function

By default, it is not possible to send ether to a function because the function will by default revert the transaction. The behavior is intentional, it should prevent Ether that is accidentally sent from being lost. However, sometimes it is necessary to pay a smart contract, e.g. in case of an ICO. Therefore, Solidity implements so-called *payable* functions.

Example

```
function buyInICO() public payable { /* … */ }
```

Function modifiers

Sometimes it is required to **check** whether a **certain condition** is true or false **before executing** a function. For instance, an authentication mechanism prior to the function call. **Writing code twice** makes it **harder to maintain** and prone to security vulnerabilities. Therefore, Solidity implements the concept of **modifiers** which are basically **a reusable piece of code**.

Modifiers are **defined** with the **keyword** modifier:

```
contract owned {
    address public owner;

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

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

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

Example for function modifiers

Internally, the actual function body is injected where _ is placed in the modifier.

The **code snippets** below **are equal**.

```
contract owned {
    address public owner;

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

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

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

===

```
contract owned {
    address public owner;

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

    function kill() public {
        require(msg.sender == owner);
        selfdestruct(owner);
    }
}
```

Chaining of function modifiers

It is **possible** to **apply multiple modifiers to a function**. The modifiers will be resolved sequentially, starting from left to right. In the example below, a user can only call the kill function if he/she is the owner of the contract and has an account balance with more than 1337 ETH.

```
contract owned {
    address public owner;

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

    modifier onlyOwner {
        require(msg.sender == owner);
        _; // Actual function code is injected here
    }

    modifier isRich {
        require(msg.sender.balance > 1337 ether);
        _; // Actual function code is injected here
    }

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

Function overloading

Solidity allows to overload functions, i.e. to define the same function twice with a different signature.
This can be helpful if a method needs to be adapted to certain situations.

Example

```
function sendEther(uint amount) {
  require(this.balance >= amount);
  msg.sender.transfer(amount);
}

function sendEther(uint amount, address to) {
  require(this.balance >= amount);
  to.transfer(amount);
}
```

If `sendEther()` is called without the address argument, the Ether will be sent to the caller.
Otherwise, it will be sent to the address passed as parameter to the function.

Named function calls

Usually, function parameters are passed by their defined signature order. **Solidity supports** the concept of **named calls. The named calls principle allows to pass function parameters explicitly via a dictionary.**

Example 1, without a named call:

```
function myAddFunction(uint a, uint b) returns (uint result) {
  return a+b;
}

function fourPlusTwo() returns (uint result) {
  return myAddFunction(4, 2);
}
```

The order of the parameters is defined by the function's signature.

Example 2, using a named call:

```
function myAddFunction(uint a, uint b) returns (uint result) {
  return a+b;
}

function fourPlusTwo() returns (uint result) {
  return myAddFunction({b: 2, a:4});
}
```

The function fourPlusTwo passes a dictionary with keys that match the signature of myAddFunction. The order within the dictionary does not matter.

Inheritance

Solidity supports inheritance of contracts. Technically, the solidity **compiler copies the code** from the **parent** contract **to the sub contract** and creates a single piece of bytecode which is deployed on the blockchain.

Solidity also **supports multiple inheritance** for a contract. In this case, the compiler just copies all parent contracts together and creates a single contract that is compiled to bytecode and deployed to the blockchain. Once a contract is deployed, it is not possible to detect from the bytecode whether a contract made use of inheritance or not.

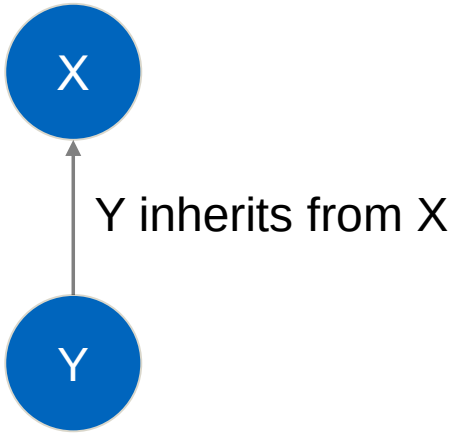
If a parent contract contains a function that is also present in the sub contract, then the functions are overloaded. In case both functions have the same signature, the sub contract's function will override the parent's function. However, the parent function can still be explicitly accessed using the `super` keyword.

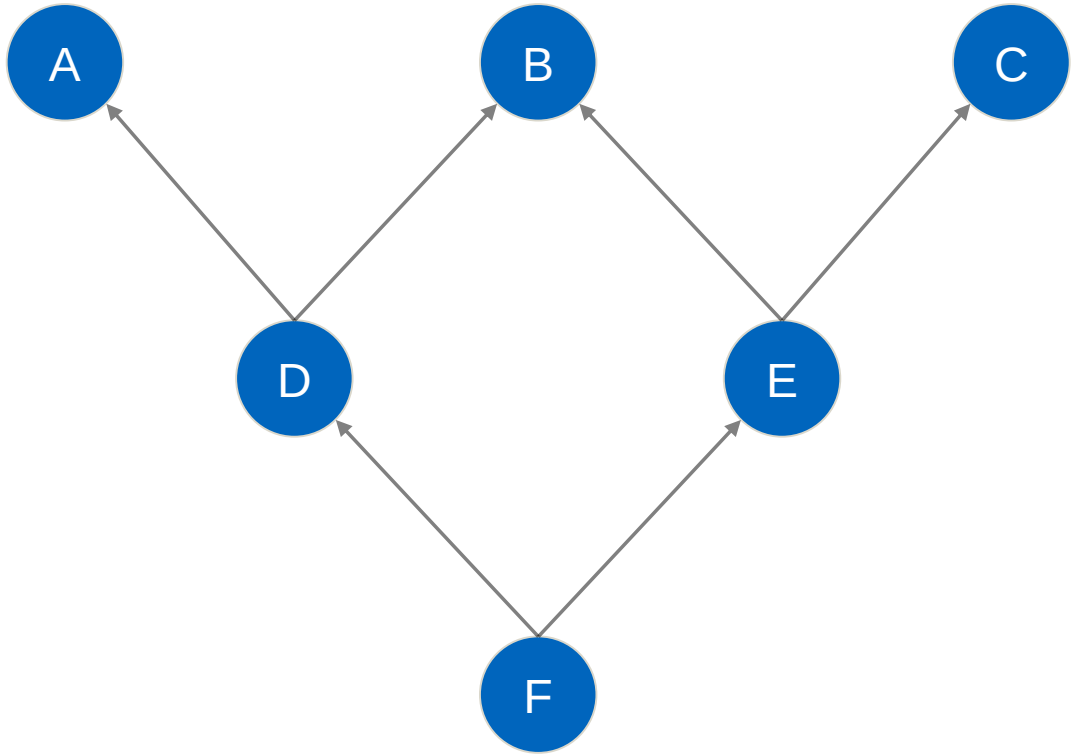
Use Cases:
SafeMath, Authentification

Multiple inheritance

Solidity uses, similar to Python, the **C3 superclass linearization** algorithm to define the order of the inherited functions. There is no implicit order for parent classes, the order is defined by the developer.

Assume the following inheritance graph





Multiple inheritance (cont.)

In Solidity, contracts can inherit from other contracts by using the keyword ***is.***

Example

```
contract A {}
contract B {}
contract C {}
contract D is A, B {}
contract E is B, C {}
contract F is D, E {}
```

The function resolution order (FRO) of the example above would be:

F, D, E, A, B, C

The keyword `super` always references the next contract in the FRO. If super would be called in `F`, it would reference to `D` and `super` in `D` would reference to `E` and so on.

Example: Inheritance

```
contract A {
 function getNumber() returns (uint a) {
  return 1337;
 }
}

contract B is A {
 function getNumber() returns (uint a) {
  return super.getNumber() + 1;
 }
}

contract C is A {
 function getNumber() returns (uint a) {
  return super.getNumber() + 2;
 }
}
contract Final is C, B {
 function getNumber() returns (uint a) {
  return super.getNumber();
 }
}
```

What would happen if Final.getNumber() is called?

- The function resolution order is:
  Final, C, B, A
  - In Final super will be resolved to C
  - In C super will point to B **!!!!**
  - In B super will point to A
- The final result is 1337 + 1 + 2 = 1340

Abstract contracts

Solidity supports abstract contracts. A contract is **implicitly** declared as **abstract**, if **one or more functions** are abstract. A function is considered abstract when it **does not have a body**.

Example

```
contract CarInsurance {
   function payMonthlyFee() returns (boolean result);
}
```

Abstract contracts cannot be compiled to bytecode. A **contract that inherits** from an abstract contract **must implement and override all methods** from the base contract to be compliable.

Abstract contracts offer a way to decouple the definition of a contract from its actual implementation. This provides better extensibility and maintainability, in particular for larger contracts.

Interfaces

Solidity supports the definition of **interfaces** for smart contracts. An interface is similar to an abstract class but is more restrictive. It is **not allowed** to define a **constructor, variables, structs and enums** in an interface. Furthermore, interfaces **cannot inherit from a contract** or **implement another interface**.

Example

```
interface CarInsurance {
   function payMonthlyFee() returns (boolean result);
}
```

A **contract** can **implement multiple interfaces** at once. According to the Ethereum foundation, some of the restrictions for interfaces might be lifted or changed in the future.

Outline

1. Solidity Introduction
   - Definition
   - Anatomy of a smart contract
   - Language features
   - Functions
   - Modifiers
   - Inheritance
   - Abstract contracts and interfaces

2. Designing Smart Contracts
   - Problem assessment
   - Modelling entities
   - Modelling transactions

3. Cross-contract and blockchain interaction
   - EVM contract function execution
   - Transactions and messages
   - Address class
   - Message object
   - Block object
   - Transaction object

Problem assessment

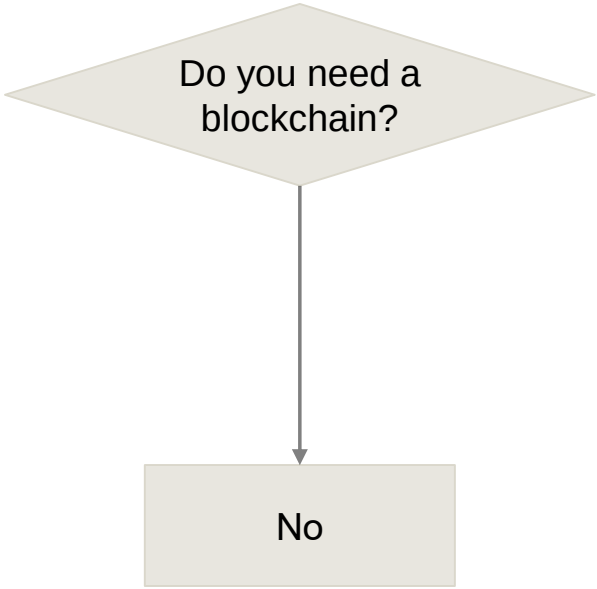
The hype of blockchain has led to a large number of software applications using blockchain technology. Often used by startups or driven by innovation departments in enterprises.

However, the real-world adoption of smart contracts still is negligible. Currently, the largest dApp using an Ethereum smart contract has less than ~11000 transactions per day (https://dappradar.com/rankings/protocol/ethereum).

Finding a use case

- Understand the problem domain and the blockchain technology well
  - Blockchain might be a potential solution if:
    - Multiple parties are involved **and**
    - The parties do not trust each other or have different interests **and**
    - Shared write access is required **and**
    - All writes to the database need to be (publicly) verifiable

- Evaluate alternative solutions.



(Joke) Model by Dave Birch (https://twitter.com/dgwbirch?lang=de)

Model the business process

As in traditional software engineering, the first step for creating an application is to model the business process.

Identification

- Identify the involved parties, systems, and their relationship.
- Identify the necessary interactions between the parties and systems.
- Identify the information that is exchanged between the parties and systems.
- Identify the system boundaries.
- Diagrams can help to get the big picture.

Modelling

- Derive concrete models from the identified parties and systems.
- Define concrete messages that are exchanged between the systems and parties.
- Define a concrete data model used by the system.
- Derive concrete interfaces for interaction with the systems and parties.
- The overall architecture is usually modeled at a high abstraction level using architecture diagrams.
- The concrete software is modeled at a lower abstraction level using class diagrams.

Fictional Example: Blood Donation

The DRK (“Deutsches Rotes Kreuz”) wants to digitize the blood donation process and make it more transparent. Therefore, it analyzes different technological solutions. As a first step, the DRK wants to track the supply chain from the blood extraction to the transfusion. The overall process is listed below.

Current simplified process:

- Person goes to the DRK and donates blood
- Blood is analyzed in laboratory and labeled
- Blood product is sent to hospital
- Hospital checks if a patients blood is compatible with the product
- Blood is transfused

Involved Systems:

- Laboratory management system (LMS) in the laboratory that analyzes the blood.
- The hospital information system (HIS) used in the hospital to manage patients and processes.

Involved Parties:

- DRK – Extracts the blood from the donor
- Laboratory – Analyzes the blood and creates blood products
- Donor – Donates his/her blood
- Hospital – Transfuses the blood
- Patient – Receives the blood transfusion

Interactions:

- DRK extracts blood from Donor
- DRK sends blood to laboratory
- Laboratory analyzes blood
- Hospital sends request to laboratory
- Laboratory sends blood to hospital
- Hospital transfuses blood to patient

Example: Blood Donation (Fictional) cont.

Assessing Blockchain and smart contracts as potential solution architecture.

Multiple parties are involved ✔

- At least four parties are involved

The parties do not trust each other or have different interests –

- Questionable, some patients might not trust the DRK and therefore do not donate.

Shared write access is required ✔

- DRK needs to read and write
- The laboratory needs to read and write
- The hospital needs to read and write

All writes to the database need to be publicly verifiable ✔

- The donor needs to be able to track what happened with his/her donation.
- In case of an accident, the blockchain could help to identify the root cause and the responsible party.

Example: Blood donation entities

In the Ethereum ecosystem two kinds of entities exist, externally owned accounts (EOAs) and contracts. Transactions are always issued by EOAs and usually controlled by an individual or a party. If an entity needs to be interactive and provide some on-chain functionality it is a candidate for a contract.

EOAs

Donor – Person who donates blood identified by their unique wallet address

DRK – Institution that runs blood donation events

Laboratory – Laboratory that takes the blood donation and creates blood products out of it

Hospital – The institution that transfuses the blood of the donor to a patient

Patient – The patient who gets a blood transfusion

Contracts

Blood Donation – Contract owned by the DRK, it tracks the time and date of the donation. Additionally, it records whenever the donation is passed to another party, e.g., from the DRK to the laboratory.

Example: Blood donation transactions

In the Ethereum ecosystem transactions and messages are the only way for entities to interact.

Interactions

DRK → CONTRACT_CREATION:

Creates a blood donation (BD) and deploys it on the blockchain**.** The address of the donor is passed via constructor and unchangeable.

DRK → BD.sendToLaboratory(address laboratory):

The DRK issues a transaction to the BD when the blood donation is sent to the laboratory. The state variable for the laboratory is set.

Laboratory → BD.sendToHospital(address hospital):

The Laboratory issues a transaction to the BD when the blood donation is sent to the hospital. The state variable for the hospital is set. Only the hospital can do the transfusion.

Hospital → BD.transfuse(address patient):

The hospital issues a transaction to the BD when the blood is transfused to a patient. A transfusion can only happen once.

Example: Blood donation EOAs

DRK: 0x91A0639dDe409c126f058e33D743b1253738C8b9

PK: 0x3aae751e36ddffd4f7d5ff4bee409583a54df823111a30f780c18cd73ebb02f8

Laboratory: 0x3aDDBa6E0C56EE1357Bb9796b20480880cA37E81

PK: 0xdbe7d4d5460f6a6e086579a0acf071b652b6ed5ae0374d704a949cbb0b740a65

Hospital: 0x582FFFacdBFDaF1936672886035ea561FF669a44

PK: 0x8ae80121c7bc29a51eb44017549280510 63a8dace9d35496dd26d0c4a1a0640c

Patient: 0x0780aFf9177d78E86Fc03158D504652f88c4D1bc

PK: 0x0a18f4e53a62e97b613ed94d0f411de327e0d1a0d5533c685042cae420aacbfb

Donor: 0x39bc67dBb1f5203AF048699233b29Dec903389A4

PK: 0x5aed62bff0a98533345482fb91ac80388869e6bdd5ad53c19b54a37468a5cb2d

Example: Blood donation contract

```
// File: BloodDonation.sol
pragma solidity >=0.4.22 <0.6.0;

contract BloodDonation {
  address donor;
  address drk;
  address laboratory;
  address hospital;
  address patient;
  bool isTransfused = false;
  modifier onlyDRK() {
        require(msg.sender == drk);_;
  }
  modifier onlyLaboratory() {
        require(msg.sender == laboratory);_;
  }
  modifier canTransfuse() {
        require(!isTransfused);
        require(msg.sender = hospital);
        _;
  }

// ends here ...
```

```
// ... continues here

  constructor(address _donor) public {
        drk = msg.sender;
        donor = _donor;
        }
  function sendToLaboratory(address _laboratory) onlyDRK {
        laboratory = _laboratory;
  }

  function sendToHospital(address _hospital) onlyLaboratory {
        hospital = _hospital;
  }

  function transfuse(address _patient) canTransfuse {
        patient = _patient;
        isTransfused = true;
  }
}
```

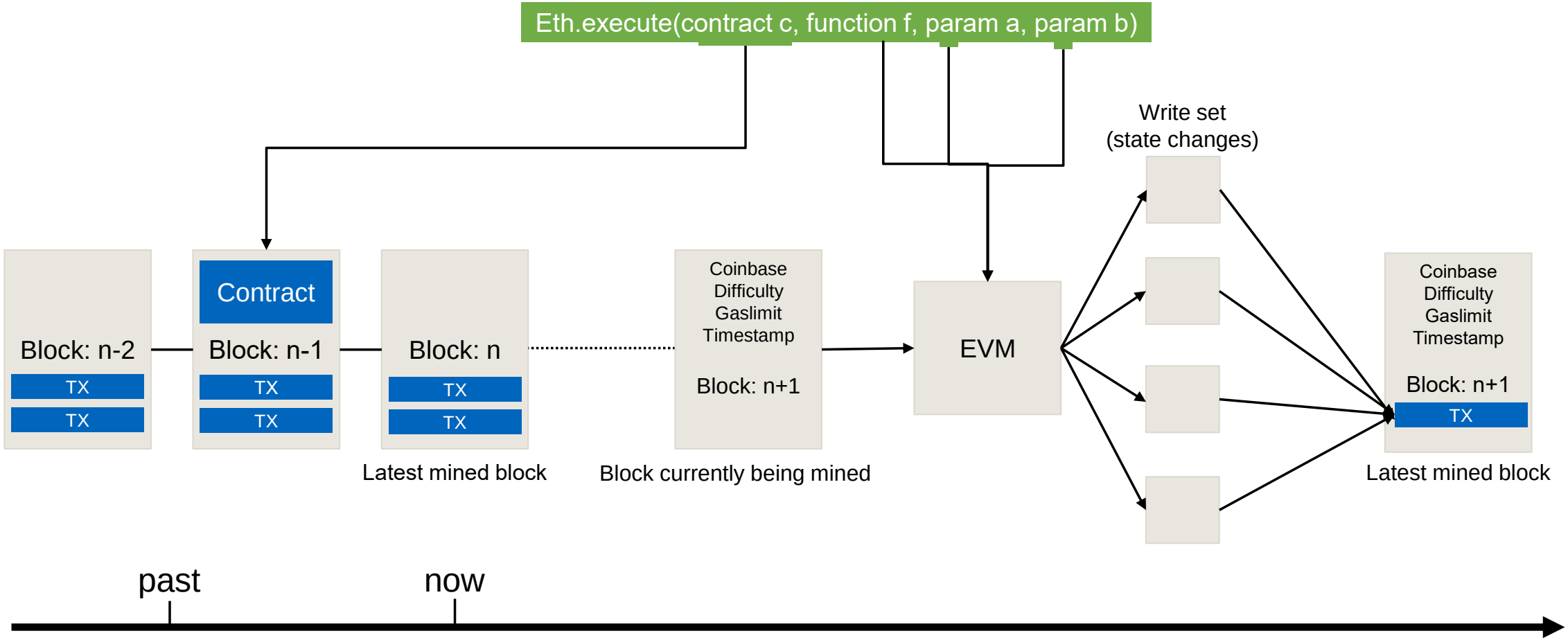
Outline

1. Solidity Introduction
   - Definition
   - Anatomy of a smart contract
   - Language features
   - Functions
   - Modifiers
   - Inheritance
   - Abstract contracts and interfaces

2. Designing Smart Contracts
   - Problem assessment
   - Modelling entities
   - Modelling transactions

3. Cross-contract and blockchain interaction
   - EVM contract function execution
   - Transactions and messages
   - Address class
   - Message object
   - Block object
   - Transaction object

Overview of EVM contract function execution

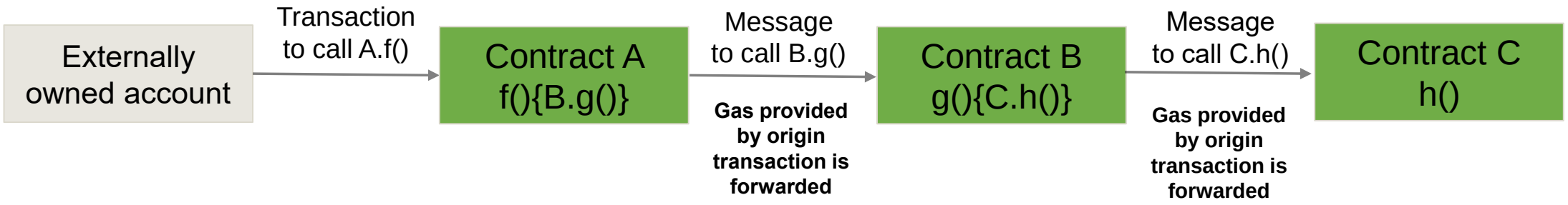


Transactions and messages

The **origin** of each **smart contract function call** is **always** a **transaction** by an externally owned account.

In more complex systems, multiple contracts communicate with each other. For example, when a contract uses an oracle contract to get information from the outside world. In such cases, the issuer of the function call must provide enough gas that also the oracle request can be fulfilled.

Whenever a contract issues a message to another contract, the gas from the origin transaction is just forwarded. However, sometimes this is not intended, e.g., when only Ether should be transferred. Therefore, the Solidity address class implements functions specifically for that use case.



Address class

Some contracts may require information about a specific account, e.g. the current account balance. Solidity implements a special type for accounts called *address*. Any Ethereum account, i.e. externally owned, as well as, contract, can be represented as address object.

An address can be directly defined via a valid 20 byte hex code representation.

```
address a = 0xd5e7726990fD197005Aae8b3f973e7f2A65b4c18
```

Furthermore, any contract object can be explicitly casted to an address.

```
contract A {
   function f() {}
}

contract B {
      function g() {
         A a = new A();
         address contract_a = address(a);
         address self = address(this);
      }
}
```

Working with addresses

It is also possible to down-cast an address to a contract:

```
A a = A(0xd5e7726990fD197005Aae8b3f973e7f2A65b4c18)
```

This only works if the contract identified by the address is an instance of A.

```
contract A {
   function f() {}
}

contract B {
      function g() {
         A a = new A();
         address contract_a = address(a);
         address self = address(this);
            // B b = B(self) would work
            // B b = B(contract_a) would fail
      }
}
```

Address class

<address>.balance
The balance of the address in Wei returned as 256 bit unsigned integer

<address>.transfer(uint256 value)
Transfers the amount passed as *value* in Wei to the *<address>*. The function throws on failure.
Forwards 2300 gas to *<address>*.

<address>.send(uint256 value)
Same as *<address>.transfer(uint256 value)* but returns false on failure

<address>.call(...)
Low level function that can be used to invoke functions but also to send Ether. The function returns false on failure and, by default, forwards all gas to <address> **(DANGEROUS!)**

<address>.delegatecall(...)
A low level function used to a call a function at *<address>* in the context / state of the current contract. This function returns false on failure.

Message object

Some contracts may require information about the caller of a function, e.g. for authentication purposes. Solidity provides the global `msg` object that contains information about the caller. It does not matter whether the caller of the function was a externally owned account or another smart contract.

The object refers to the last account that was responsible for invoking the function. This can either be a contract or an externally owned account.

`msg.sender`
The account address the function's caller of type `address`

`msg.data`
The complete payload of the message / transaction

`msg.sig`
The function's hash signature so that the EVM knows which function is called

`msg.value`
The amount of Wei that is sent with the message

Message object

Since the message object always refers to the last sender, it requires some special attention when used in combination with this in a contract.

```
contract A {
   function f() public returns (address a) {
      return msg.sender;
   }

   function g() public returns (address a) {
      return f(); // f() is called directly msg.sender will be the address which calls g(). f() does not need          to be public
   }

   function h() public returns (address a) {
      return this.f(); // f() is called by the current contract instance => msg.sender will always be equal
         // to address(this). f() has to be public
   }
}
```

Block object

Some contracts may require information about the latest mined block, e.g. when a specific function should be time locked. Solidity provides a global variable called `block` to access the most recent block of the blockchain.

`block.coinbase`
The account address of the current block's miner

`block.difficulty`
The current mining difficulty as unsigned integer

`block.gaslimit`
The current block's gaslimit (by the miner)

`block.timestamp`
The UNIX timestamp of the block (in theory, can by manipulated by the miner)

Transaction object

The global tx is similar to the msg object and provides information about the transaction that triggered the function call.

The main difference is that tx always refers to a transaction, i.e. it's source is always an externally owned account.

tx.origin
The issuer of the transaction. This is always an externally owned account.
DO NOT USE FOR AUTHENTICATION!

tx.gasprice
Information about the gas price that was used by the issuer of the transaction