Software Engineering Conference Russia 2019

November 14-16
St. Petersburg

Agile Software Development Automated by Blockchain Smart Contracts

M. Marchesi, M.I. Lunesu, M. Ortu, A. Pinna, R. Tonelli,
University of Cagliari, Italy

Giuseppe Destefanis, Brunel University London, UK

Valentina Lenarduzzi, Tampere Univ. of Technology, Finland

Blockchain

- The Blockchain was a technology originally devised to run the Bitcoin cryptocurrency in a decentralized and secure way

- It is a distributed data structure characterized by:
  - data redundancy
  - check of transaction requirements before validation
  - recording of transactions in sequentially ordered blocks
  - ownership based on public-key cryptography
  - immutability
  - a transaction scripting language, associated to the transactions – the corresponding program is executed by all nodes

Blockchain for Agile Development

We exploit blockchain technology for Agile Software Development for:

- improving productivity
- automating tasks
- integrating tests
- leveraging Product Owner duties
- agreeing upon conditions
- automating payments

Blockchain for Agile Development

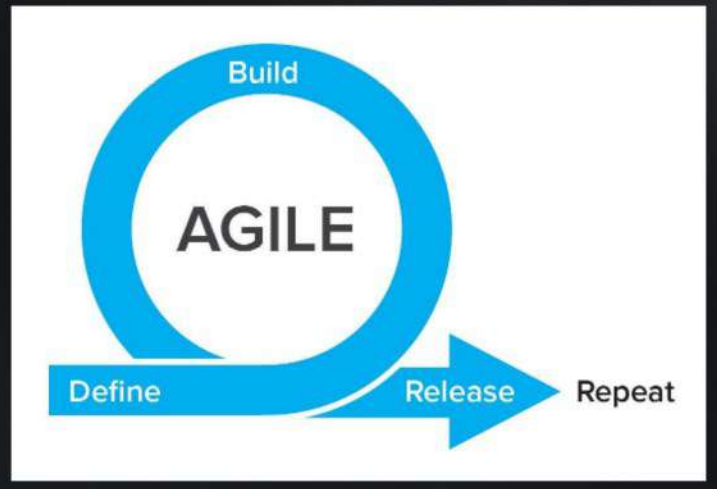
- Novel approach to Feature-Driven development
- Blockchain provides:

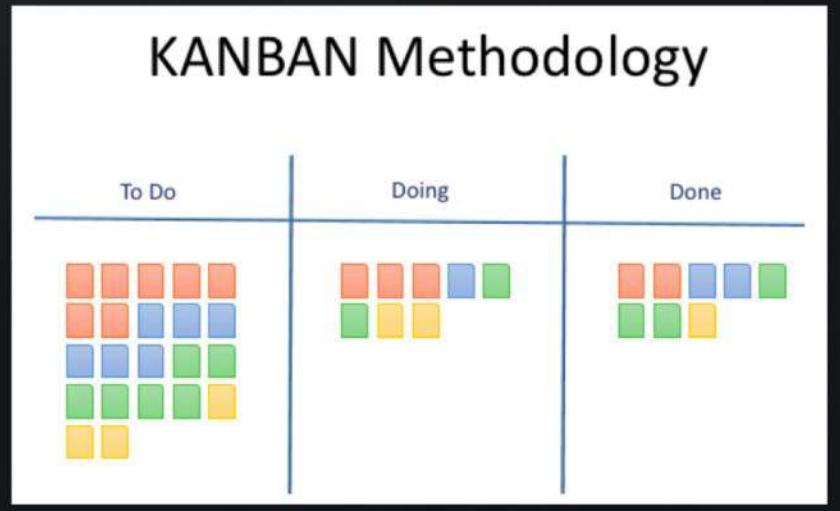
  transparency, identification, non repudiability, traceability, encryption, ...
- Smart Contracts provide:

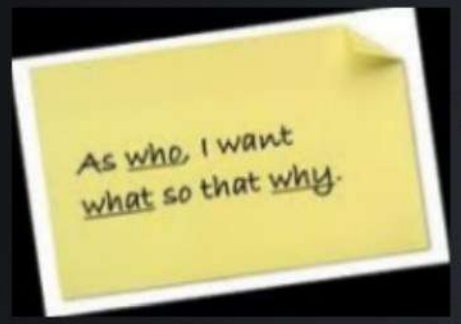
  clear conditions for stakeholders, check for operations and conditions, automated test checking, automated payments, ...

Agile Methodologies

- Focus on the ability to respond to change quickly and easily.
- Most popular AM is Scrum, followed by Scrum/XP hybrid, Scrumban, Kanban and others.
- These software development methods offer adaptive planning, early delivery and continuous improvement.







User Stories and Acceptance Tests

- In AMs system requirements are expressed through atomic **increments** (Minimum Marketable Features - MMF) or User Stories (USs)
- USs explicitly describe an interaction among the system to develop and external actors.
- The development process **is guided** by these increments.
- Each US is typically provided of one or more Acceptance Tests (ATs), describing specific scenarios of usage, giving actual state and inputs, and required output data.

US and Acceptance Tests (2)

- Iterative AMs, such as Scrum, advance by **short iterations** (Sprints), assigning to each iteration a set of USs to implement.
- AMs based on **continuous flow**, such as Kanban, proceed by implementing USs through a continuous flow, aided by the Kanban board.
- In both cases, ATs are crucial to decide if a US has been correctly implemented or not, and thus its developer(s) can be compensated.

The Role of Product Owner

- In Scrum, the Product Owner (PO), is the key role for managing, prioritizing and explaining USs to the development team.
- The PO is in charge of verifying the implementation of USs and executing the related ATs.
- If the PO agrees that a US is correctly implemented, the US is marked as "done" and can be paid by the customer.
- If automated ATs related to a US pass, the PO gives (possibly after a further check) the green light to the compensation of the team, or of the developer, for completing the US.

Smart Contracts and Blockchain

- Smart Contracts (SC) represent well defined programs running oon a blockchain, implementing simple and autonomous tasks
- SC ususally have a well defined purpose and cannot be modified, they provide services to the Contract callers (or to contract parties)
- SC are deployed into a Blockchain framework, and their source code **can be checked by all parties**
- Blockchain features natively implement secure and transparent access to SCs

SCs and Blockchain (2)

- Smart Contracts interactions occurs by mean of **messages**, sent through blockchain transactions
- Transactions may be used to execute Smart Contracts code, or to perform payments in the Blockchain cryptocurrency
- Users are identified by blockchain addresses, and can be other SCs as well
- Smart Contracts can identify the sender of the message, and execute code with different policies settings for different users

We propose a system for:

- Relieving the duties of the PO for certifying the correctness of the process outcome, delegating them to Smart Contracts written in Solidity language and deployed on the Ethereum Blockchain
- Automating some specific tasks
- Asseverating and validating Acceptance Tests
- Performing payment in Ether cryptocurrency, based on the agreements and on correct US implementation

Main ideas

- US-driven development is a **contract** between the customer and the developer team, supervised by the PO
- A US is considered **completed** (and thus should be paid) **when its ATs pass**, and the PO asseverates the completion
- ATs can (and should) be automated, so that:
  - AT inputs and initial system state can be precisely defined
  - The prescribed result of AT execution can be uniquely defined
- Developers must not know in advance the ATs and their result (to avoid opportunistic behavior)
  - However, they can know in advance the hash digest of the result

Main ideas (2)

- SCs on a blockchain (Ethereum) can be used to manage the process
- They are transparent, and can be checked by all parties
- Payment can be made by a SC using Ethers or tokens having a given monetary value
- It is duty of the customer to provide Ethers or tokens to the SCs
- The PO manages the system, entering the addresses (identity) of the developers, the USs and their ATs, with the hash of their results
- The PO assigns the USs to developers (or these subscribe to USs)
- Once a US is completed the developer runs its ATs, computes the results and their hash digests, and sends them to the SC

Main ideas (3)

- If the two hashes overlap for all ATs, the SC can directly pay the developer, or the final payment may require a final PO's approval
- If the developer fails to provide the correct hash digest of an AT, the US is not considered done. S/he is allowed to provide it later
- The SCs code cannot be changed and transactions registered in the blocks can trace timestamp and all operations performed by SCs
- All transactions in the Blockchain public ledger can easily be verified and are immediately available providing full transparency to all process actors

System architecture

- We used ABCDE – Agile BlockChain Dapp Engineering for architecture design (presented in a talk at SECR 2018):
  - UML diagrams and scenarios for describing the system are used to represent both SCs and App parts of the system
- The application we propose is composed by:

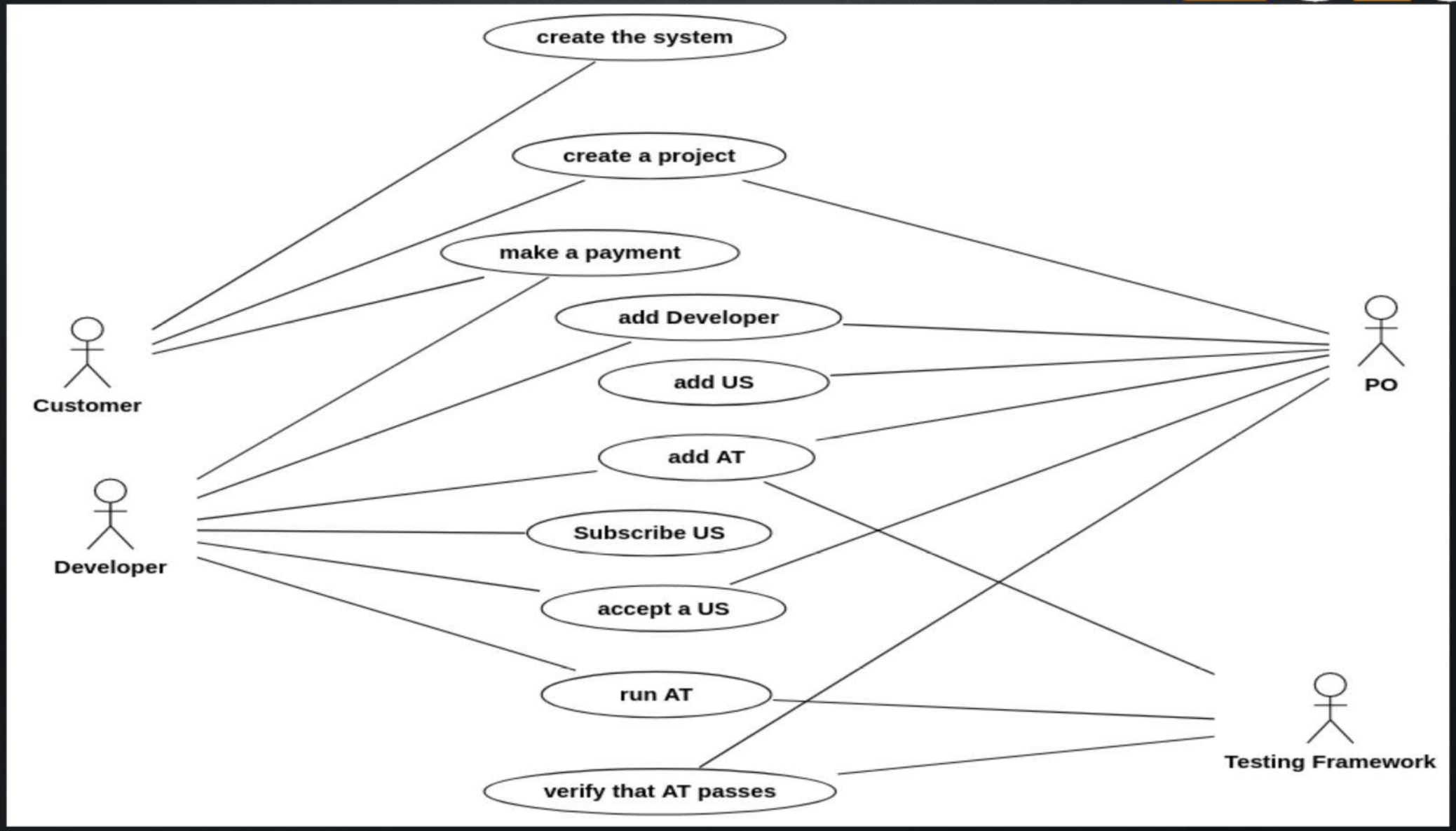
  1) a traditional software system, running on servers/mobile devices, communicating with users and external devices;

  2) the SCs running on the Blockchain.

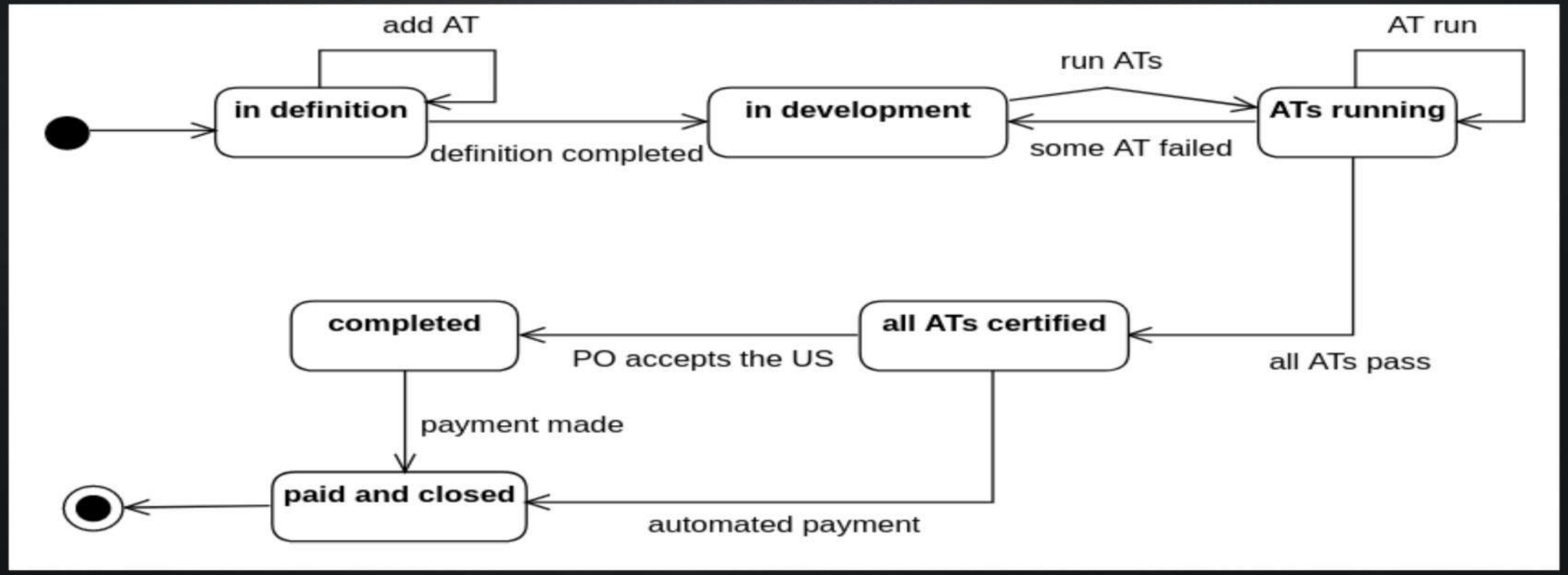
Actors of the System

- **Customer**: s/he owns the SCs implementing the system, enables the PO and the team members, specify the rules for automated compensation, provides Ethers or tokens
- **Product Owner**: s/he specifies and manages the USs and the ATs, including the correct inputs and outputs of the Ats. S/he certifies the correct implementation of a US, if needed.
- **Developer**: a team member who writes the US and related ATs, and is entitled to be paid for her/his work.
- **Testing Framework**: an external system enabling writing and execution of ATs, and the verification that they pass.

Use Case Diagram



Evolution of the state of a US



- A US is created, and is provided with one or more ATs, whose correct result can be coded in a file, whose hash digest is written by the PO in the blockchain, using her/his cryptographic credentials.

Evolution of the state of a US

- The developers do not know the content of the file with AT results, but only its hash digest.

  → they cannot fake the AT result, by directly writing the expected result in the AT code

- USs completely implemented by a developer are committed and the related ATs are run
- When the tests pass, they generate a result file which must be identical to the original one, and hence with the same hash digest.
- This triggers the run of SC code on the blockchain, which computes and compares the two hashes, certifying the US validation if they are the same
- The hashes of AT results can be sent to the SC by the developer, or automatically by the test suite

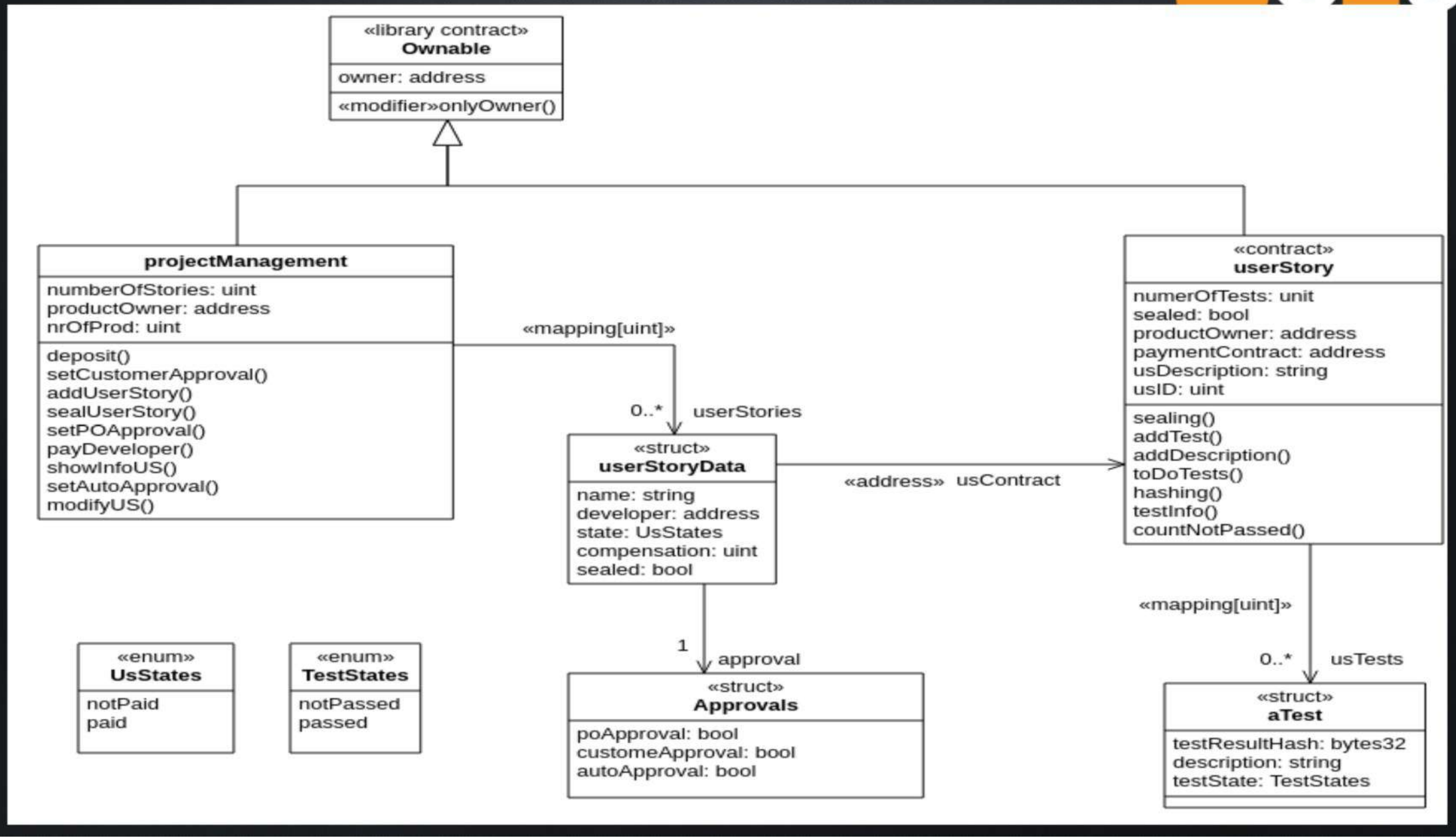
The Blockchain System

- We implemented the requirements using two SCs, "projectManagement" and "userStory"
- ProjectManagement represents a software project whose USs have to be developed, tested and eventually accepted.
- This contract can be deployed and owned by the Customer, who can appoint the PO, and can deposit an Ether amount to pay the compensation of developers who create USs whose ATs pass.
- Using this contract, the PO can add USs, seal US, and approve the USs s/he deems are completed.

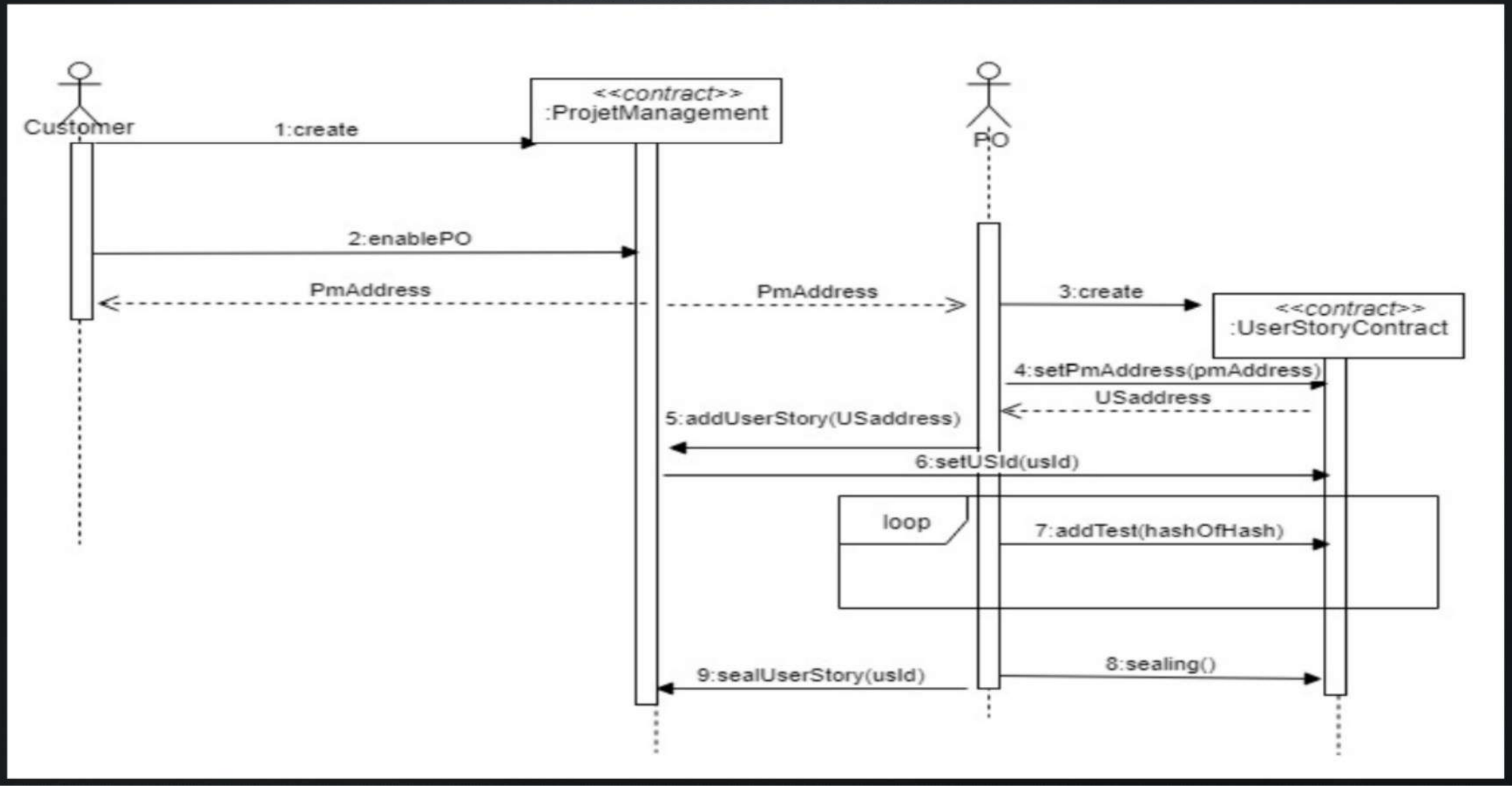
The Blockchain System

- UserStory: a new contract userStory is created by the PO for each US of the managed project with the description of the US and the list of its ATs
- The Developer can access the list of not yet validated USs tests and read the information related to the US and its ATs
- For each test, s/he can submit the hash code of the result of the test, to certify it passed
- The SCs projectManagement and userStory are linked to each other: the PO deploys a userStory SC whenever s/he needs to add a new US to a given project.
- After the deployment, the PO will add the new US to the projectManagement SC, which will able to access the US contract through its Ethereum address.

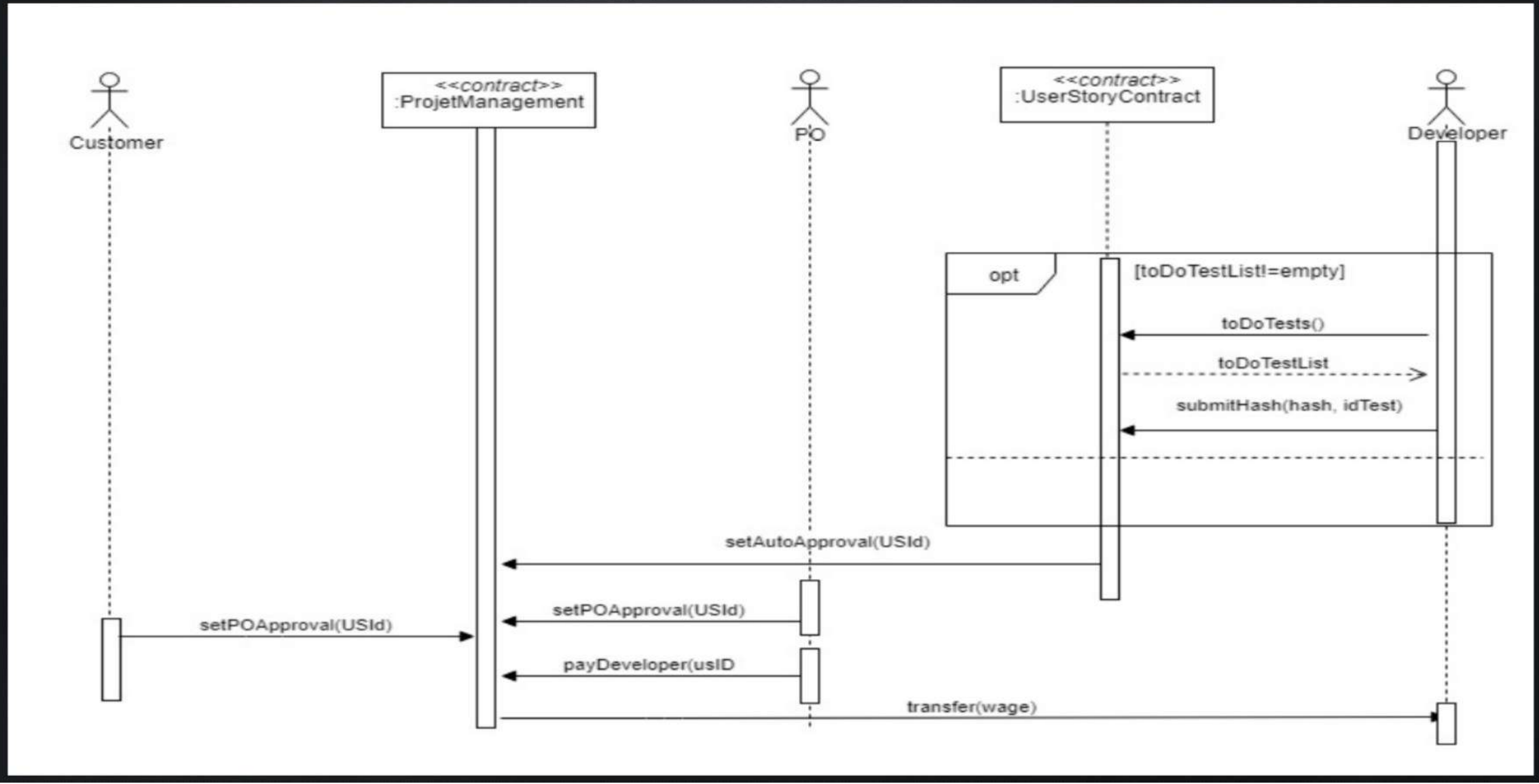
The UML modified class diagram



UML Sequence diagram representing the definition of a US



UML Sequence diagram representing the completion and acceptance of a US.



Implementation

- We adopted the Behavior-Driven-Development (BDD) Agile Methodology to define, implement and check the automated Acceptance Tests
- We implemented the SCs to verify the passing of the test related to a given user story and used to transfer the agreed amount of cryptocurrency to the blockchain address of the developers team.
- The prototype has three main components:
- - the Smart Contract deployed on an Ethereum testnet

  - a Web application to be used by the Product Owner

  - the BDD framework used by developers

Implementation

- The Product Owner meets the team to agree on a set of user Stories to develop and on the related Acceptance Tests (ATs)
- The AT must be verified according to the BDD scheme and on the chosen test automation tool
- Each AT is written by the PO into a Smart Contract with the correct answer masked by an hashing fingerprint and deployed on the blockchain.
- For each developed User Story, the team implements one or more ATs using the BDD framework agreed upon with the PO

Implementation

- When a test is executed, the team, provided with a blockchain address and blockchain credentials, accesses the ABI exposed by the smart contract and sends the evaluated answer through a blockchain transaction
- The SC hashes the team's blockchain message and automatically checks the hash against the value provided by the Product Owner
- The SC can send back the answer to the AT, which passes only if the hashes match

Implementation

- Let us consider the following simple feature expressed using BDD structure:

```
Feature: Simple maths
  In order to do maths
  As a developer
  I want to increment variables

  Scenario: easy maths
    Given a variable set to 1
    When I increment the variable by 1
    Then the variable should contain 2

  Scenario Outline: much more complex stuff
    Given a variable set to <var>
    When I increment the variable by <increment>
    Then the variable should contain <result>

    Examples:
      | var | increment | result |
      | 100 |         5 |    105 |
      |  99 |      1234 |   1333 |
      |  12 |         5 |     17 |
```

Implementation

- This Feature can be directly coded using Cucumber.js framework:

```
1  // features/support/world.js
2  const { setWorldConstructor } = require('cucumber')
3
4  class CustomWorld {
5    constructor() {
6      this.variable = 0
7    }
8
9    setTo(number) {
10     this.variable = number
11   }
12
13   incrementBy(number) {
14     this.variable += number
15   }
16 }
17
18 setWorldConstructor(CustomWorld)
```

Implementation

- CustomWorld is the class under test. The Given, When, Then clauses described in the Feature ”Simple maths” are directly coded in a file named steps.js.
- Cucumber.js executes the matching of what is written in natural language in the Feature and what is coded in the steps.js file.

```
1  // features/support/steps.js
2  const { Given, When, Then } = require('cucumber')
3  const { expect } = require('chai')
4  Given('a variable set to {int}', function(number) {
5    this.setTo(number)
6  })
7
8  When('I increment the variable by {int}', function(number
       ) {
9    this.incrementBy(number)
10 })
11 Then('the variable should contain {int}', function(number
       ) {
12   expect(this.variable).to.eql(number)
13 })
```

Implementation

- The framework automatically matches the Feature written in natural language (the clauses Given, Then and When) and the code implemented in Javascript.
- The first scenario contains one assertion (the variable should contain 2), hence this generates one AT
- The second scenario contains a table with three matches (the variable should contain <result>) using the placeholder <result> for the values in the table.
- The second scenario generate three ATs.
- Once the ATs are expressed using BDD, the test failure or success can be automatically checked by a Smart Contracts.

Implementation: part of SC code

```
contract userStory{
    ...
    enum TestStates {notPassed, passed}

    struct atest{  //to be setted by product Owner
        bytes32  testResultHash; //hash of the hashcode
     of test result
        string description;
        TestStates testState;
    }

    mapping(uint => atest ) usTests;

    //global variable
    uint numberofTests;
    bool sealed = false;
    address productOwner;
    address paymentContract;
    string public usDescription = "Pleas add a
     description";
    uint usID=0;

    constructor(address pmAddress) public{
        productOwner = msg.sender;
        paymentContract = pmAddress;
    }
    ...
    ...
     //core function called by a developer to try if the
     test is passed
    //The developer put in input the hash of its results
     and the related test ID
    function submitHash(bytes32 myHash, uint idTest)
     ifSealed ifNotPassed(idTest) public {...}
    function getPaymentContract() view public returns(
     address){...}
    function getusID() view public returns(uint
     userStoryID){...}
    ...
    ...
```

Results

- We used our prototype to execute a case study where PO, Customer and Developers team agreed on executing a set of ATs on the USs of software produced using Scrum methodology.
- The PO inserted the hashes of the correct expected results for the acceptance tests into SCs deployed in the Ropsten Ethereum test
- These SCs are managed by a second SC which was used as a master of the first one, allowing the counting of the passed tests and the management of the reward in cryptocurrency.
- The team had a Ropsten Ethereum account related to a blockchain address provided with the amount of Ethers needed to perform the blockchain transactions for checking the ATs on the SCs

Results

- Depending on the AT results sent with a message by the team, the SC answered a return value reporting the failure or success of the test.
- After the team successfully completed all the tests of a US, they were automatically rewarded by the agreed amount of cryptocurrency in their own Ropsten blockchain address
- The PO asseverated the payment by the master SC once all the acceptance tests had passed

Conclusion

- The results show that our prototype could easily be used to leverage part of PO duties in a Scrum, or a Lean-Kanban agile process
- Several parts of the process, from management and verification of acceptance tests, to the insertion of new user stories, to the final payment of development team can be automated
- The native nature of the blockchain as a trustless technology perfectly suites the spirit of mutual trust of Agile Manifesto, since the blockchain can “create trust”, allowing different parties to make a transaction “without relying on trust”