Università degli Studi di Pisa

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

SCOOP: A user-friendly framework for Smart Contract prOtOtyPing

Candidato:

Saverio Catania

Relatori:

Prof(.ssa) Laura Emilia Maria Ricci

Dr. Damiano Di Francesco Maesa

Anno Accademico 2020-2021

Contents

Chapter 1

Introduction

In the last two decades blockchain systems have emerged and thrived. Several improvements of Bitcoin - the first blockchain technology for tracking digital tokens - have been proposed in the last years and we begin to see the first non crytpocurrency related applications.

Blockchains allow to record digital information in a tamper-proof, transparent and chronological way. The information is distributed among the computers composing the blockchain network - namely peers - who maintain the blockchain. As peers are mutually untrusted and trust the protocol rather than each other, the blockchain is often called a trust-less technology.

There is a lot of interest in this technology from two main fronts: academia and industry. Both parties are particularly interested in the novel blockchain frontier: smart contracts.

Smart contracts allow blockchain developers to automate the enforcement of certain kinds of agreements within the blockchain. The fields of application are numerous: from supply chain to insurance, from management of medical records to copyright protection, and so on.

When an entity takes the responsibility of using smart contracts to regulate and control real-life scenarios, testing efficiently and pragmatically smart contracts becomes extremely important. To this day, different blockchain platforms support smart contracts, although they may present very different characteristics. Deciding which blockchain platform presents the most desirable features for a given task can be a delicate matter too. Testing smart contracts against different blockchain platforms and against the purpose of the smart contract itself is therefore another task that needs to be taken into account.

In general, we refer to the process of designing and perfecting a (group of) smart contract(s) as the smart contract prototyping process. Several frameworks, tools and libraries implement this process on different blockchain

systems, often addressing very specific matters.

We argue that the degree of user-friendliness and usability of those frameworks is low and that, to the best of our knowledge, there is not enough focus on the smart contract prototyping process within the current state of the art.

SCOOP comes from this consideration, adding the fact that, while lots of users may be interested in creating smart contracts and interacting with them, the majority of them will encounter several obstacles doing it using the current technology stack, due to its specificity and level of expertise required.

Therefore, the main goal of SCOOP is to create a whole new level that inhabits between inexperienced users and the current smart contract prototyping technology, allowing the user to have a more user-friendly, smart contract-centered interaction that does not require high expertise and study on behalf of the user in order to be used. SCOOP must therefore be as generic as possible, in order to cover a variety of blockchain platforms, smart contract languages, and any technology-specific parameter, hiding those details to the user.

Not only SCOOP must be independent with respect to blockchain platforms and smart contract languages, but also in terms of the already existing frameworks that interact with these elements. In fact, since those frameworks already perform technology-specific tasks, SCOOP's role in this case will be to recycle that technology-specific behaviour enhancing the discoverability[1] degree of smart contract prototyping frameworks. We therefore focus on creating a bridge - the SCOOP's API - between the functionalities provided by a given smart contract prototyping framework and the users.

In order to motivate the design choices made while building SCOOP, we cover in Chapter 2 basic notions about software engineering, blockchain technology, and smart contracts. In Chapter 3 we study the current state of the art in terms of frameworks that manage the smart contract prototyping process. The knowledge acquired in Chapter 3 allows us to improve the quality of the assumptions SCOOP makes about smart contract prototyping frameworks in Chapter 4, i.e. the design choices, which is the core of this thesis. The design choices chapter highlights the role of the user interface (UI) and the SCOOP's API that sub-modules implement in order to be compatible with SCOOP. In Chapter 5 we propose an implementation of SCOOP's key features described in Chapter 4. Since we will only cover some aspects of SCOOP's infrastructure this chapter will provide what we call an implementation skeleton. Finally, Chapter 6 provides an example of sub-module using the technologies presented in Chapter 5. The sub-module in question

[1] An object has a good degree of discoverability when it is possible to determine what actions the user can perform with it and the current state of the object itself [1].

is created by implementing the SCOOP API reported in Chapter 4. We provide in the bibliography[2] a prototype of SCOOP that currently uses a simplified version of SCOOP's API. The thesis is concluded by Chapter 7, where we sum up our findings and highlight possible future directions to expand our work.

Chapter 2

Background

This chapter describes the ground notions of the work presented in this thesis. In particular, the concepts of software engineering, blockchain and smart contracts will be covered. In Section 2.1 we discuss the strict relationship between abstraction and, in general, we articulate some software engineering principles that have been particularly useful for the architectural design of `SCOOP`. The terms blockchain and transaction will be introduced, respectively in Sections 2.2 and 2.2.1, as these constitute the building blocks of smart contracts, described in Section 2.2.2. The difference between a smart contract and a real world contract will be highlighted in Section 2.2.3, in order to clarify what smart contracts' limitations are. These blockchain fundamentals will guide the understanding of the prototyping process relatively to a smart contract setting, as described in Section 2.2.4.

Section 2.3 provides a brief description of the new decentralized web, focusing the attention on the issues that developers have to face with an ever changing and complex architecture such as the Web3 in Section 2.3.1.

All of these concepts have been useful for understanding the state of the art and for selecting the common features among different blockchain architectures and smart contracts. In fact, the contents of this chapter helped in giving `SCOOP` the adequate level of abstraction. This consideration will be better expanded in Section 2.4, the summary of this chapter.

2.1 Software Engineering Principles

Software engineering is the systematic application of engineering approaches to the development of software. This field offers a lot of tools, techniques, principles, and theories that help focus the development on portability, maintainability, scalability, flexibility, and a lot of other characteristics of the code

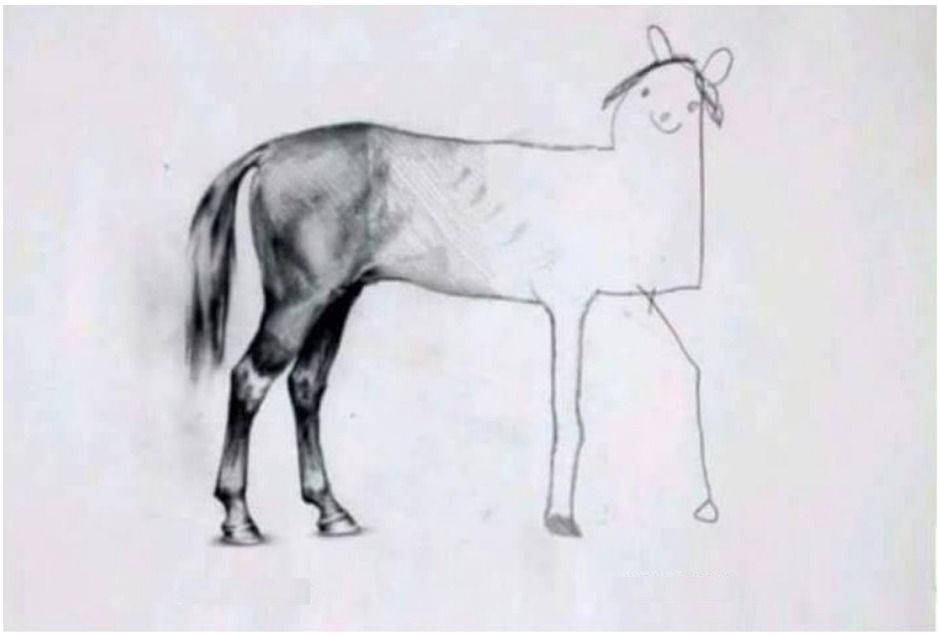

Figure 2.1: A model of a horse, with different levels of abstraction.

one is writing [3]. Solving problems related to the development of a software requires both general abilities, such as the ability of abstracting as explained in Section 2.1.1, and specific abilities, such as the specific task of modularizing a complex infrastructure described in Section 2.1.2.

2.1.1 Abstraction and Models

In a software engineering setting, the term *abstraction* refers to both the process of abstracting from irrelevant details and the result of the process itself [4]. From now on, the conceptual or physical object deriving from the abstraction process will be referred to as *model*. The abstraction level of a model is inversely proportional to the number of relevant details considered, since the more detailed a model is, the less generic it will be. Any given object can be represented at very different levels of abstraction, resulting in a set of valid models for that object, each of them withholding a variable amount of information. Figure 2.1 shows a model of a horse with different levels of abstraction, the least abstract on the left and the most on the right. The highest level of abstraction gives the information the model is a horse, but does not provide information about the race of the horse, making the horse generic. In general, specifying implies adding constraints and details whilst generalizing means removing them.

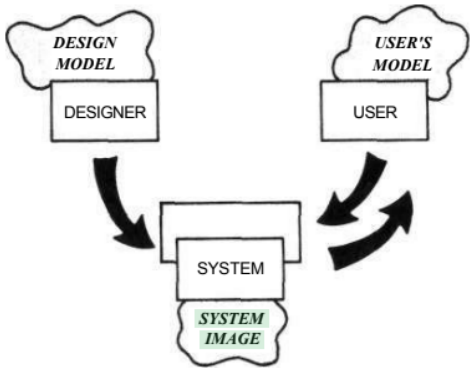

Figure 2.2: The system image, conceived by the designer's conceptual model, is distorted in the user's mental model[1].

It is worth noting that models are necessary in general, since it is often impossible to replicate exactly real existing objects. This misrepresentation of real objects is not the only one in play, there is also a distorsion in the way models are perceived by the observer, this is generally due to cultural and environmental constraints of the latter. In the interaction design field, the product of the distorsion the observer performs is called *mental model* [1].

In software engineering, the development team follows techniques whose aim is to understand which features the software should possess, as well as how to disclose them. Each member of the development team answers these two questions in different ways, thus producing a conceptual model that is different from the other members'. In turn, the final user does not possess the same knowledge base of the programmers and designers, so *he will use the software differently from what was expected by the team, basing on his mental model.

We derive that, when designing a software, the design team has to keep in mind that the same conceptual model adopted by the designer will differ from the mental model of the final user, as shown in Figure 2.2. This observation leads us to the understanding that, no matter how much modelling effort the developer puts into the design of an interface, there will always be a final user that has difficulties understanding it.

These concepts together influenced and continue to shape the advancement of `SCOOP`'s front-end and back-end design. From a User Experience (`UX`) design perspective, the focus is designing a `GUI` whose prime goal is to meet the needs of the final user, providing an interface as intuitive and simple as possible using a *Human Centered Design* approach [1]. On the opposite side, the main goal of `SCOOP`'s back-end is to provide an equally appropriate inter-

face to the modules underneath the framework itself. Since SCOOP needs to address the issue in a modular way it is necessary to learn the best practises in software engineering literature when it comes to modular applications.

2.1.2 Modularity

From a back-end perspective, the main trade-off in building SCOOP is the one between the degree of specificity of the modules' API and the number of modules whose behaviour can be modeled through the API itself. From the considerations in Section 2.1.1 we can derive that generalizing the API means removing assumptions about the modules, widening the domain of use of SCOOP and shifting responsibilities to the modules, whereas a process of specification adds constraints and therefore limits the number of supported modules while making it easier for the module developer to implement the API. Finding the balancing point in the trade-off means providing an API that is general enough to cover all the possible needs of modules without making the API itself trivial. A set of principles named SOLID [5], introduced by Barbara Liskov, has been a useful swiss army knife when it came to solve this trade-off.

One noteworthy principle is the *dependency inversion* one, which is a specific form of loosely coupling software modules. When following this principle, the conventional dependency relationships from the high level interface and the modules are reversed in such a way that the interface will not depend on the implementation details of the modules [5]. It follows that the application of this principle facilitates modular programming. An intuitive explanation of the principle is shown in Figure 2.3.

Another relevant and far more general principle is the *information hiding* one, which states that segregating the interface from the implementation is a key aspect of design that makes the resulting code secure and legible. This result also helps having a better separation of concern experienced by the ones using the interface and those who develop it.

2.2 Blockchain fundamentals

A blockchain is a particular kind of distributed ledger technology (DLT), where the ledger is a growing list of blocks, each containing information about the previous one using cryptography and therefore forming a chain of blocks. As shown in Figure 2.4, each block is hash-linked, and modifying the body of a block changes its hash, thus making it incompatible with the link stored in the following block [6]. This ensures the incorruptibility of

Figure 2.3: *Would you solder a lamp directly to the electrical wiring in a wall?*: No, create an interface (wall socket) electrical equipment have to depend on.

the ledger, because in no reasonable amount of time it would be possible to alter the content of a list of blocks of reasonable length. A blockchain is maintained by peers/nodes that exchange data with each other via a common communication protocol and agree on the order in which blocks are registered in the ledger. To validate new blocks, a node needs to join the network and adhere to the related validation protocol, this together with cryptography enables different parties of the network to trust each other. The alternative is to trust a centralized system, although the latter is far more susceptible to external attacks, as it has what we call a *single-point-of-failure* [5].

It is not mentioned in the latter definition how a peer can join or access the network, this is due to the fact that, in general, a blockchain architecture can also be classified in terms of the ways a peer can join or access it. In general, a blockchain can be classified in two orthogonal ways: private/public and permissioned/permissionless[1].

The first classification refers to reading privileges of transactions, in the sense that private blockchains only allow some authorized users to visualize

[1]This classification will come in handy later, as an example of what different parameters a module might need in order to interact with a specific deployed blockchain; for now it is sufficient to say that, for example, logging into a network might require addresses and ports known by the final user.

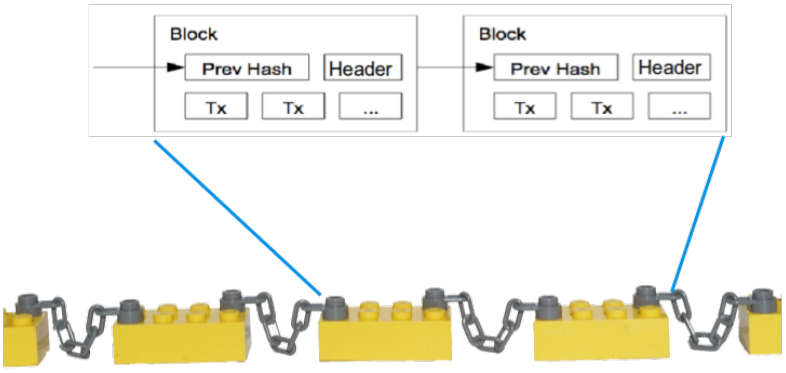

Figure 2.4: Each block stores the hash of the previous one, so that incorruptibility is guaranteed.

transaction information, whereas public blockchains do not have this constraint. In the case of permissioned/permissionless blockchains the authorization concerns the possibility of validating transactions. We derive that a particular blockchain system could be classified using any combination of these labels. An interesting case is the Ethereum protocol, it is used in the *mainnet*, which is public and permissionless, as well as in private and permissioned instances[7].

A common use case is the one in which blockchain data has to be public, but signing/maintaining transactions the network must be done by an enterprise or the state. A naive example: in a food chain setting, whoever handles the food first-hand has to be the only one authorized to confirm a successful delivery, whilst everyone has the right to verify where the food is coming from.

There are hundreds of possible blockchain systems, so it is useful to find a common model to describe their architecture. In the next chapter, the frameworks we are going to discuss provide a few valid models with very

different levels of specificity.

This concepts already let us have a grasp of how disruptive this innovation is, since it is the very first time in history in which we can store data in such a way that is almost impossible for potentially malicious groups to alter it. It is not the only distributed ledger technology, but it is broadly recognised as the most secure one. A key question that comes to mind is what to put in the distributed ledger. The idea of Satoshi Nakamoto (an unknown person or group of individuals) is dated 2008 and it is to use the ledger to store transaction information concerning digital movements of tokens, namely *bitcoins*, as well as the ownership status of each token in the network. A system of this sort is named *distributed electronic cash system*[2].

When speaking of blockchains, the electronic token in question is named *cryptocurrency*, a neologism indicating a digital medium of exchange based on cryptography. These coins' ownership and transaction records are stored in the `DLT`.

Finally, in blockchains, as well as in the majority of exchange platforms, a transaction has a cost related to the amount of energy, time, or computational work required in order to validate the transaction itself. We name *validation time* of a transaction the amount of time the blockchain network uses to process and validate the transaction.

Without demanding completeness, let us present two strictly related concepts: transactions and contracts.

2.2.1 Transactions and Contracts

Generally speaking, a *transaction* is something, given in exchange for something. In particular, contract law academics divide this "something" in two categories, promises and performances [9]. Both promises and performance include what we call the "object" of the promise/performance itself. For instance, the promise "I will sing for you in September first" includes the object of the promise, which is the singing. In a *performance*, such as "I will sing for you now", the object of the transaction (still the singing) is delivered to the counterpart immediately after the transaction started.

Formally, in a promise, the object is delivered following the terms and deadlines specified by the related contract, in the sense that the instant of realization of the promise does not coincide with the beginning of the transaction. Indeed, performing a transaction means engaging in a contract which has terms, terms being normally implied by a legal system and enforced

[2]This is not the first attempt to create an electronic cash system, as the first concrete contribute to this field dates back to 1982, with David Chaum conceiving *ecash* [8].

by a central entity, for example the government.

It is now clear the strong relationship between contracts and transactions, regardless of the digital or physical domain in which we place these two key elements.

An example of a decentralized contract enforcing entity is the electronic cash system Bitcoin: in this case, the implicit contract is the mechanism that regulates the exchange of money triggered by a transaction (the engaging part), which in turn produces a predictable result, that is to say, a successful bitcoin transfer. The transaction is valid if all the implied terms are satisfied: for instance, executing the transfer of an amount that the sender does not own produces - or should produce - an invalid transaction. Moreover, Bitcoin prevents an attack called *double spending* [10], which allows the attacker to successfully transfer the same tokens to more than one person.

It is worth noting that, in the Bitcoin case, there are no promises involved, since the exchange involves the transaction validation request coming from an individual and the process of validation provided by the network, with the first immediately triggering the second. This particular kind of transactions are therefore classified as *performance-for-performance*[3][9].

2.2.2 Smart Contracts

The implicit contract described by the Bitcoin protocol, and its counterparts, is a real breakthrough that helps bringing unconditional trust between potentially suspicious parties in electronic financial environments, as it represents the first real solution to the double-spending problem [10]. On the other hand, it is difficult for developers to associate custom behavior to a transaction in the Bitcoin case since it only provides a Turing-incomplete scripting language that can express simple dynamics.

A solution to this problem has been the introduction of *smart contracts*, defined as computer programs written using Turing-complete languages and running on `DLT`s, which model the automation of legal or interesting aspects of a contract or agreement.

Smart contracts encapsulate behavior described by code that dictates the terms of the smart contract. Actions performed in this environment correspond to transactions. Plus, the contract can have a memory that represents its state and it has a balance, so it is possible to send and receive

[3]We can compare this sort of transaction to the one between an individual and a vending machine, here there are no promises involved since the moment of selection of a product coincides with the start of the mechanical arm. In this analogy, the performance of the mechanical arm represents the computational work performed by the blockchain network in the validation process

tokens to the latter. With these characteristics, a smart contract becomes very similar to an individual, which is able to exchange a service for money.

For some blockchain systems, a smart contract is a particular kind of *account*, which is generally associated with an individual. In some other cases, such as *HyperLedger*, there exist only one type of account named *chaincode*. An example of smart contract written in the language Solidity for the Ethereum blockchain can be found in Listing 2.1.

```
contract SimpleStorage {
    uint storedData;

    function setValue(uint val) public {
        storedData = x;
    }

    function getValue() public view returns (uint) {
        return storedData;
    }
}
```

Listing 2.1: An example of smart contract, written in Solidity.

This contract's state is a integer number, in any given instant there's just one value associated with *storedData* and anyone that has access to the network can modify (via the method *setValue*) or visualize (via *getValue*) this value. Every update happens with the approval of the network in a secure way and, most importantly, history of the modifications is accessible publicly. An important aspect of smart contracts concerns their execution. In general, when compiling a smart contract, the output is a bytecode, this is what will actually be run on the blockchain. The bytecode needs to be interpreted, this task is performed in general using a virtual machine. In the Ethereum case, we have the Ethereum Virtual Machine (`EVM`). When a user triggers a transaction containing a smart contract method call, the latter is expressed in bytecode too. Since developers use high-level languages to make method calls, the need of translating those high-level calls into bytecode emerges. Performing this task is possible thanks to the `ABI` (Application Binary Interface), which also takes care of the opposite process, thus the one that translates the low-level response coming from the virtual machine to the high-level one, that is delivered to the source of the call. It is worth noticing that `ABI`s are often expressed in a well-known format, so that new languages and frameworks that adopt that format can trust the `EVM` to execute the

smart contract's byte code correctly.

To bring an example, let us show in Listing 2.2 the `ABI` of the "SimpleStorage" contract.

```
[
    {
      "constant": true,
      "inputs": [],
      "name": "value",
      "outputs": [
        {
          "name": "",
          "type": "uint256"
        }
      ],
      "payable": false,
      "stateMutability": "view",
      "type": "function"
    },
    {
      "inputs": [],
      "payable": false,
      "stateMutability": "nonpayable",
      "type": "constructor"
    },
    {
      "constant": false,
      "inputs": [
        {
          "name": "val",
          "type": "uint256"
        }
      ],
      "name": "setValue",
      "outputs": [],
      "payable": false,
      "stateMutability": "nonpayable",
      "type": "function"
    },
    {
      "constant": false,
```

```
        "inputs": [],
        "name": "getValue",
        "outputs": [
          {
            "name": "",
            "type": "uint256"
          }
        ],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    }
]
```

Listing 2.2: The `ABI` of the "SimpleStorage" smart contract.

In this case, the `ABI` is in `JSON` format and consists in an array of objects, each object corresponding to a method of the smart contract. Of particular interest are the fields:

- **name**: String that corresponds to the name of the method.
- **inputs**: Object that describes names and types of the method's parameters.
- **outputs**: Object that describes the type of the return value.
- **type**: Field that corresponds to the type of the method, such as "function" or "constructor".

2.2.3 Limitations

With the introduction of smart contracts, contract terms can be specified by code. One question to ask is what limits these smart contracts have, as well as to what extent they can emulate a traditional system, such as the legal one. A way of answering this is by stating that the objective of contract law is not to eliminate risk, but rather to allocate risk. If one had to eliminate risks from legal binding contracts, it would be necessary to specify every possible situation that could happen. This procedure has a cost and often cannot be applied de facto. In contract law, this issue is addressed by introducing ambiguities in situations that have a lesser probability of happening, which represents the allocation of risk. An unlikely situation of this sort, namely

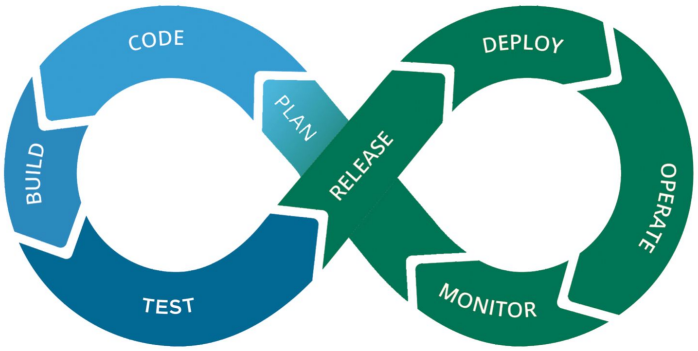

Figure 2.5: The DevOps cycle. It is a Software Development Lifecycle (SDLC) methodology where each stage shapes the execution of the next one. SCOOP, together with the sub-modules, aims to assist the user in every step of the cycle[12].

breach, is eventually processed by a court, which in this setting represents a break processing system [9].

We derive that code cannot substitute or be separated from the law and the legal system, partially because not every contract expressed in natural language can be modelled by a smart contract due to the presence of ambiguity, which code cannot handle.

From a computer science perspective, we can now state that a contract written in machine code can produce reliable and deterministic results only inside a restricted - and therefore limited - domain, within which ambiguities are not allowed. This is the reason why smart contracts are also being referred to as *dumb contracts* in contract law[9].

2.2.4 Smart Contract Prototyping

By definition, prototyping is an experimental process where design teams implement ideas into tangible forms [11]. In modern computer science, the DevOps cycle presented in Figure 2.5 can help us break down the prototyping process of complex software.

If the idea in question is to design smart contracts, by substitution we obtain the definition of *smart contract prototyping*. More specifically, prototyping a smart contract comprehends a set of platform-specific tasks, although

in general it is possible to individuate some structure inside this process. To bring an example: in every case a smart contract will need to be edited, more dynamic tasks could include its *deployment*, sending transactions to it, and *testing* its functionalities. The term deploy in this context is used to refer to the publication of the smart contract to a particular network, as it happens for the publication of the network itself. The testing of a smart contract can be a delicate matter since a deployed contract cannot be modified and therefore patched when a bug has been found. Also, different testing methodologies have different levels of complexity. In order to understand intuitively why, three types of testing paradigms will be discussed: black box test, unit test and interaction test. [13]

The *black box test* treats the contract as an input/output circuit device, providing inputs and analyzing from the outside the outputs as well as metrics such as computational time and energy consumption.

Unit testing adds to it the possibility of looking inside the smart contract's state, thus providing more information.

The *interaction testing* is a more complex and dynamic technique since it involves two or more contracts, potentially making the system under test a chaotic one.

The tools that allow us to perform all of the above-mentioned activities share some characteristics too, to individuate those common features is part of the problem that `SCOOP` aims to solve.

2.3 Decentralized Web

Blockchain technologies are evolving at an incredible speed, including the frameworks that surround them. They are actually part of a bigger picture, the *Web3*, which is a promising attempt of creating a new communication web on a global scale based upon peer-to-peer communication [14]. Applications running on the Web3 are called `DApps`, which stands for *Distributed Applications*. In particular, a `Dapp` can either run on a blockchain or on any sort of decentralized peer-to-peer network, an immediate implication is that smart contracts can be considered `DApps`.

One can notice from the definition that the Web3 has an heterogeneous structure, in fact it lies upon different types of peer-to-peer networks, so there is no standardized procedure for inexperienced users to adopt that facilitates access to it. There are some concrete attempts by some individuals and groups such as the Web3 Foundation, whose work is having an important impact in defining what the decentralized web could become.

2.3.1 Stratification

In complex IT infrastructures, one can often find a layered architecture. This tendency can be justified thinking about the software engineering principles discussed previously. One classic example is the `TCP/IP` stack, the most used set of data transfer protocols used to access the Internet as we know it today.

In this sort of structures, every level encapsulates some information about the overlying one[15], with the most high level - the first one - being the one the majority of participants interacts with. That first level must have a user-friendly interface.

This mechanism is repeating itself on the application level in the decentralized web environment, as well as in all the tools that surround it. In a blockchain and smart contract setting, we can find various tools and `APIs` that rely on more raw ones in order to work properly, especially when engaging with smart contracts. There is a continue advancement and research in all of these fields, and even the smart contracts' languages are still in a rough form.

It is a fact that this continuous change of the Web3 stack makes it particularly hard to build and maintain distributed applications or even frameworks who depend upon the above mentioned `APIs`. This consideration had a huge impact in the design process of `SCOOP`, since it caused the majority of the design choices to be made in an technology-invariant perspective.

Generally speaking, it is difficult for a developer to interact with blockchains, suffice it to say that specific activities such as deploying smart contracts or testing them still require a pretty large knowledge base about the underlying stack. This collides with the previously mentioned information hiding principle, one of the key elements that favours progress in technology.

2.4 Summary

Let us summarize the discussed matters, trying to pinpoint and anticipate some of the critical points that `SCOOP` has to cope with and that will be better faced in Chapter 4.

The interface this framework provides must have the right level of abstraction towards the sub-modules that use it, this means finding the most specific model that works for any possible sub-module. Hence a sub-module is, in general, considered as a component that exposes functionalities corresponding to a prototyping process. This generalization work ought to be performed in respect of the different kinds of blockchain platforms, smart contract languages, and frameworks that interact with them.

The interface also needs to make as few assumptions as possible on how the sub-modules underneath process user's inputs and store their own state. In addition, the interface must not alter the submodules' behavior and has to make it fairly simple to attach a sub-module to the interface itself.

From a user standpoint, `SCOOP` must have a friendly `UI` and be transparent in terms of the results of the requested actions. Moreover, the interface should enable the user to access all of the functionalities of the underlying module.

In order to build `SCOOP`, these issues must be addressed. In general, the resolution of each critical point has come down to a trade-off or a design choice that opened up some possibilities whilst excluding others, hence many solutions will be presented. We will find some solutions to these trade-offs in already existing frameworks in Chapter 3.

`SCOOP` is one of a few in its main goal, in fact it has the expressive power to use those other frameworks as sub-modules, so the actual state of the art and this framework are not mutually exclusive tools and therefore are not considered as competitors.

Chapter 3

State of the Art

In this chapter, we will present a review of the main frameworks available for blockchain benchmarking and smart contract prototyping. In particular, four frameworks have been chosen, each addressing a different problem and/or working at a different level of abstraction. Understanding the state of the art lets us avoid reinventing the wheel and permits to compare already existing solutions to the `SCOOP` proposition, which will be described in the next chapter.

There has been an ever growing number of articles that refer the term "blockchain" in the last years, we can see that interest in this technology is growing accordingly by both public and private entities. This trend can be recognised also in the large number of frameworks that provide any type of blockchain-interaction automation, including smart contract prototyping.

The problem with those frameworks is that many of them present a command-line interface, complex configuration steps, dense documentation and/or a high level of expertise. `SCOOP` instead aims provide the tools to solve these problems.

We are going to discuss some key examples of frameworks focusing on the automated process in question as well as the platforms they support, keeping in mind that `SCOOP`'s goal is to maximize the adherence to these two different aspects across different sub-modules. These considerations will show how different tools managed to partially address the issues mentioned in the previous chapter, such as structuring the prototyping process and finding the right level of abstraction when building an interface.

3.1 Frameworks

The following is an overview of the frameworks we choose to analyse, in order of complexity and number of supported blockchain platforms:

- **Truffle Suite:** Set of tools to automate the smart contract prototyping process, mainly in platforms adopting an Ethereum Virtual Machine (`EVM`)[16].
- **Blockbench:** Focuses the attention exclusively on comparing the performance metrics of private blockchain technologies which are already deployed[17].
- **HyperLedger Caliper:** Another performance analysis tool that adds deploying management, portability to new testbeds, and supports mainly HyperLedger systems[18].
- **BCTMark:** It is considered the most complete framework for blockchain performance analysis. Supports every blockchain and can perform resource reservation[19].

It is worth noting that these frameworks solve different problems at different levels of abstraction. We are going to discuss the first framework, which gives a concrete idea of what prototyping a smart contract means in an `EVM` environment, and then we'll show the evolution of blockchain benchmarking with the remaining frameworks. In this last scenario, some aspects of this evolution will be highlighted, in particular: the number of supported blockchain systems, degree of portability, and characteristics of the smart contract prototyping process. Of particular interest for our purposes will be the design choices of these frameworks.

As anticipated in Section 2.1.1, the less general a framework is in terms of supported platforms, the more specific are the functionalities it provides.

3.1.1 Truffle Suite

Truffle Suite gets developers from idea to `DApp` as comfortably as possible. We can notice that this description fits well the definition of prototyping process stated in Section 2.2.4. Truffle Suite consists in three frameworks (Truffle[20], Ganache[21] and Drizzle[22]), each enabling the user to handle a different aspect of the `DevOps` process for smart contracts. It is used by both novices and big enterprises, resulting in one of the most successful attempts to build a set of tools to manage and monitor smart contracts' life-cycle[23]. The main tools we are going to discuss are *Truffle* and *Ganache*, although

it is worth mentioning *Drizzle*, which provides several libraries that aim to help developers build better front ends for their `DApps`.

Truffle

It is the main tool of the suite and provides an intuitive command-line interface. It works only with blockchain platforms that rely on an Ethereum Virtual Machine and HyperLedger `EVM`, its permissioned version. The initialization on Truffle on a given directory produces an empty Truffle project, although some prepared examples are provided. Given the restricted domain of interest, Truffle is able to provide a lot of interesting features for smart contract developers, provides features such as:

- **Deployment automation**: in Truffle-terms, we would say that the developer can write a migration script that helps describing, in a synthetic way, which order smart contracts have to be deployed.
- **An API for contract abstractions**: accessible through the package "truffle-contract". It is possible to know which properties and methods a smart contract exposes, as well as interacting with one or more of its deployed versions.
- **Easy configuration**: via a single configuration file, in a few rows one can describe which provider will elaborate our remote procedure calls, which network our contracts are going to be deployed to and other information, such as the compiler version.
- **Simple commands and an interactive console**: for example, a basic deployment can be performed using two commands, one for compiling the source code and the other to migrate to the desired network using the migration script.

Some other features like smart contract testing and debugging are offered, although the interesting aspects for our purpose concerns mainly the points above.

To visualize the general workflow of Truffle, let us comment Figure 3.1.

Users create a new Truffle project adding their smart contracts' source code to it. When the user compiles the contract using the "compile" command, the output includes the contract's `ABI` and bytecode. These two elements are necessary to handle the depoloying of the smart contract, that the user performs using the "deploy" or "migrate" command.

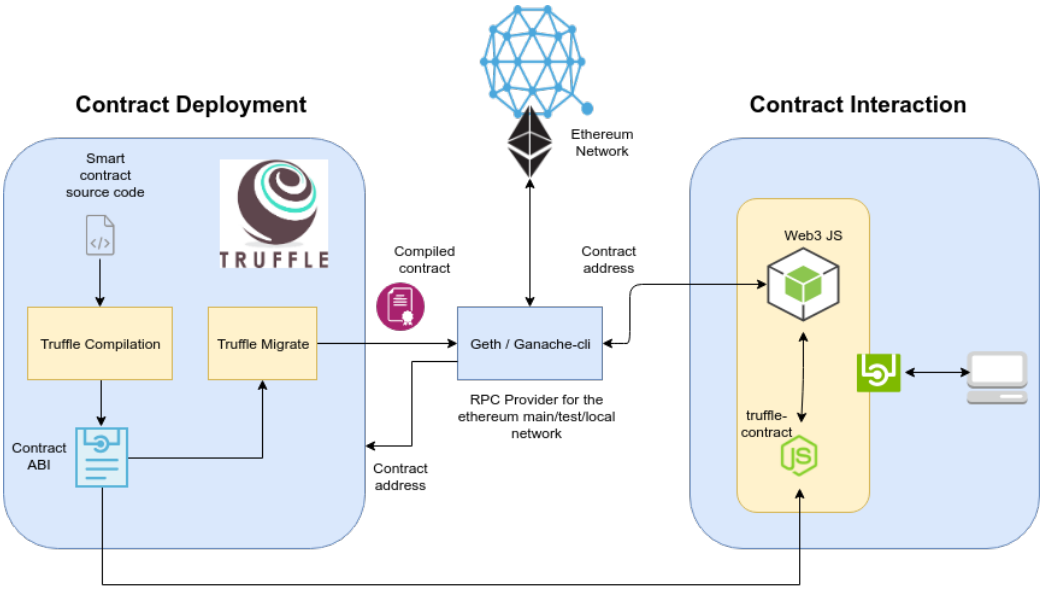

Figure 3.1: This diagram shows how Truffle's main components interact with each other, as well as with other external entities such as the remote access point(s) that allows accessing the ethereum network. There is also a distinction between the two stages of deployment and interaction (of/with a smart contract).

The deployment takes place in a local environment or using Remote Procedure Calls (`RPCs`), in this last case Truffle needs the user to insert information to contact the entity that provides such an `RPC` system (`RPC` provider).

In both local and remote deployments, the output, for each contract, is the address of the deployed contract that Truffle later uses to interact with it.

To let the user easily interact with contracts *he deployed, Truffle provides a contract abstraction library compatible to all the supported blockchain platforms. The abstraction is the wrapper library "truffle-contract", which in turn uses `Web3JS`, another library for smart contract interaction that interacts with smart contracts at a lower level.

Ganache

Ganache is a "one-click blockchain" for both Ethereum and Corda distributed application environments. It is formally known as `testRPC` and comes as a command line or a graphical tool. It solves the problem of connecting to real testnets in order to test `DApps`, since this practice can slow down the advancement of the prototyping process. It does so by creating a one-node

local blockchain, by signing transactions autonomously and by providing an interface to inspect the blockchain state and other interesting features, such as linking a Truffle project to a given workspace.

For example, it is possible to:

- visualize the history of transactions from and to a given wallet address;
- monitor the events that a contract emits;
- check the contract state;
- manage independent workspaces.

3.1.2 Blockbench

Blockbench was the first tool ever addressing the problem of comparing different blockchain systems using specific quantitative metrics. It can inspect the behaviour of private blockchains and compare them, individuating bottlenecks within their infrastructure.

For example, a given blockchain system **A** can outperform **B** by any measure whilst being outperformed when the number of nodes exceeds a certain threshold. This kind of investigation would lead us to say that **A** has a lower scaling capacity than **B**.

The transaction is the atomic unit used for performance analysis, in particular it is possible to measure *throughput* and *latency*, that respectively indicate the number of successfully submitted transactions per second and the validation time per transaction. Note that these are common measures, often found in other frameworks too. Two other measures used by Blockbench are scalability and fault tolerance, both defined in terms of throughput and latency variation. The fault tolerance evaluation is performed by simulating failure at different levels of the blockchain system [17].

Figure 3.2 shows the architectural model of Blockbench with respect to the levels of the blockchain it aims to test. Although the role of each level does not directly concern our matter, it is important to notice the role of the classification itself. In fact, Blockbench abstracts this information from all the possible blockchain systems, associating a *workload* to each level, defined as any kind of data used to test the respective level. For example, at the application level a workload could be a smart contract that stores key-value pairs, together with the storage initialization values and a set of methods that read/write a certain number of entries, in order to stress-test the smart contract itself.

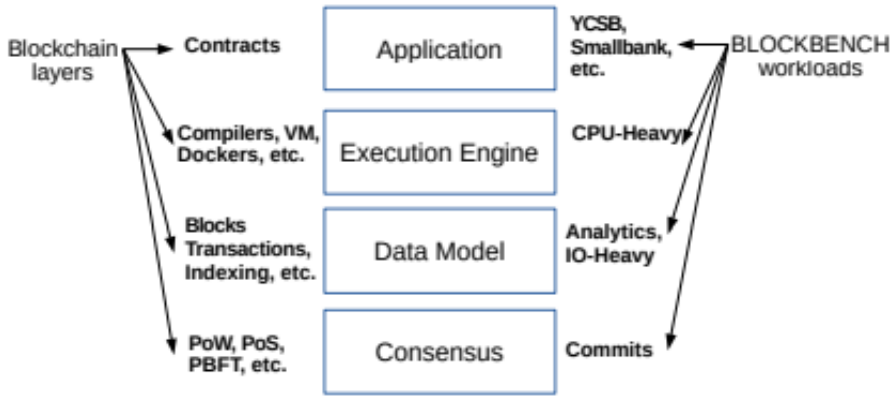

Figure 3.2: Layered structure of the blockchain used by the Blockbench team. Each level is tested using a different, ad-hoc designed workload [17].

Furthermore, in order to integrate a new blockchain system, Blockbench provides an `API` that needs to be implemented.

It follows that Blockbench is really useful when the necessity is to find bottlenecks between adjacent levels and to compare levels of different blockchains. The key point is that one has to deploy the blockchain System Under Test (`SUT`), as well as to implement the given `API` in order to use this framework. This work has been done by the development team of Blockbench on Ethereum, Parity, and HyperLedger platforms, as they implemented the interfaces for workloads, demonstrating some inherent properties of each platform[17].

It becomes now clear that, using Blockbench, benchmarks must be designed ad-hoc for each layer of a specific `SUT` and therefore they're not portable to different testbeds. Moreover, it lacks the most important feature in smart contract prototyping, the deployment management of the smart contract themselves.

3.1.3 HyperLedger Caliper

This framework supports blockchain platforms such as Ethereum, HyperLedger Besu and HyperLedger Fabric. It is transaction-based and tests blockchain platforms' performances generating as output a "benchmark report", it can easily be used to test smart contracts' functionalities. Particularly of interest is the architecture: since a lot of design challenges faced by `SCOOP` have also been found in Caliper documentation, this framework's features are going to be discussed more in depth. Some relevant aspects that separate this tool from Blockcbench are portability to new testbeds and de-

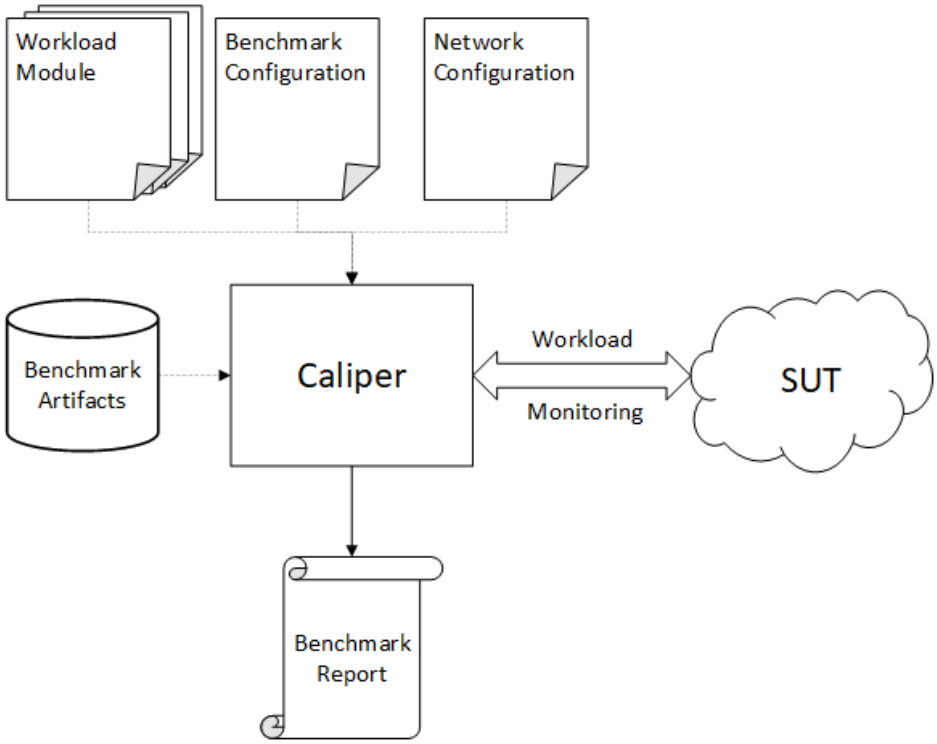

Figure 3.3: Bird's eye view architecture of Caliper.[18]

ployment management. For now, let us go through the general architecture of Caliper, shown in figure 3.3.

Artifacts are, as is Blockbench, any important data in support of the prototyping and benchmarking process such as smart contracts' code and environment setup scripts, with the latter being executed before or after the benchmarking process. In contrast to Blockbench, Caliper does not use a leveled structure of the `SUT` in order to perform benchmarks. Instead, it abstracts from different platforms by individuating operations they may have in common. This is done by providing an `API` that exposes methods to be implemented in order to:

- Initialize the `SUT`.
- Install the contracts.
- Monitor the `SUT`'s state.
- Submit a transaction.

In this context, a component that implements these functionalities is called `SUT` Adapter and its behaviour is `SUT`-specific. Two important artifacts of this

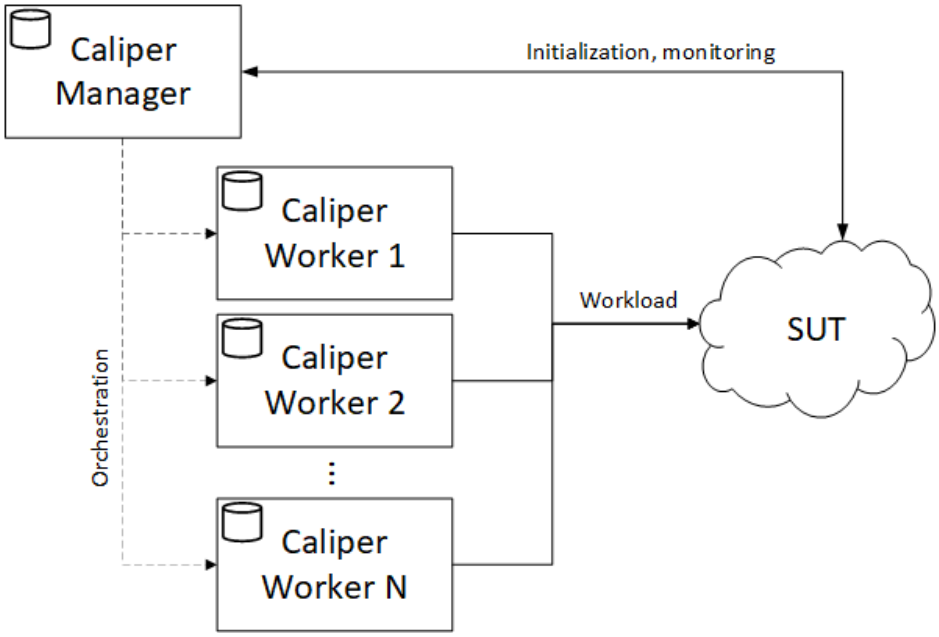

Figure 3.4: Relationship between Caliper Manager and Workers [18].

framework are the "network configuration" and "benchmark configuration" files. The first is another `SUT`-specific component that provides metadata about the platform, such as the network topology or the IP addresses of each node, the second consists in a more dynamic structure that describes how the benchmark process will be performed at a high level. In particular, the benchmark configuration file is to be considered the flow orchestrator of the benchmark, which is structured in rounds, each of which associated with a workload module, the transaction generation rate for that round and other metadata. The workload module has the task of generating transactions for a specific round, the output of this process is a valid transaction for the specific `SUT` and it will be submitted by invoking the `SUT`'s Adapter.

The most important architectural choice regards the separation of concern between the flow orchestration and the actual performance benchmarking, respectively performed by the "Caliper Manager" and the "Caliper Workers" as the schema in Figure 3.4 shows. The Caliper Manager is responsible for scheduling rounds and it asynchronously enables workers to submit transactions prepared by round-specific workload modules.

In Figure 3.5 we can see the high-level representation of the Caliper manager process, As we can see, the `SUT` initialization and monitoring tasks are left to the manager, which also schedules the rounds described in the benchmark configuration file. The Caliper Manager work remains the same when

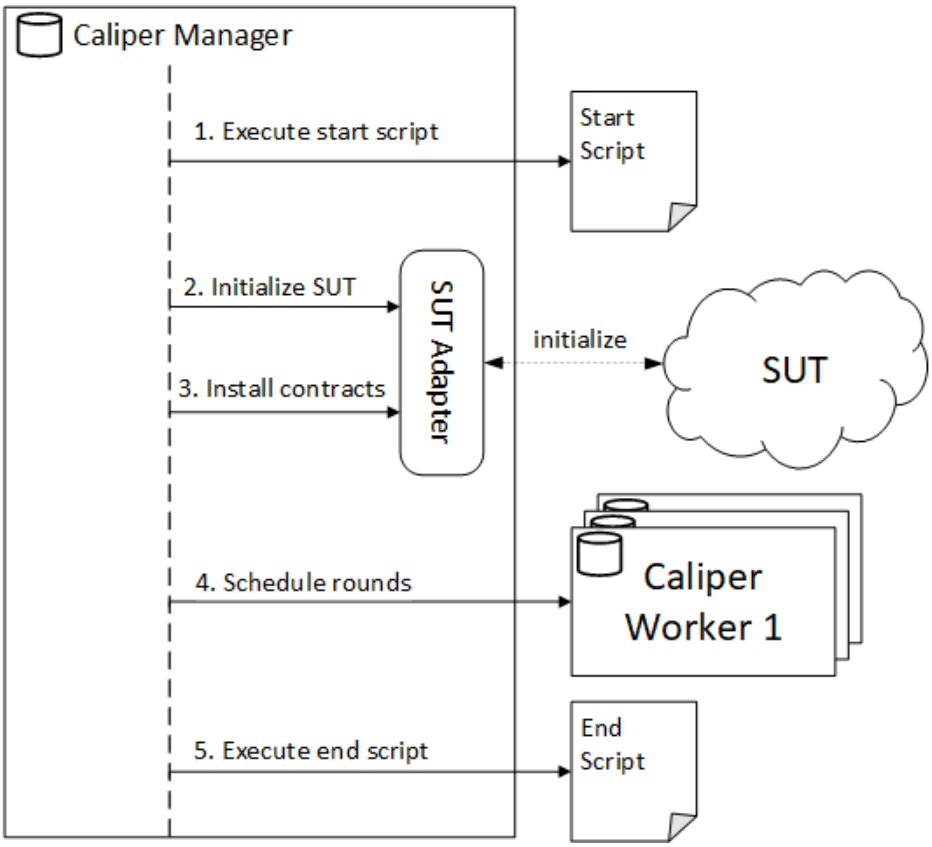

Figure 3.5: Caliper main manager process.

considering different SUTs, in opposition to the Caliper Workers, whose behaviour is specifically related to the workload module that generates the transaction and the SUT Adapter that receives it.

From the last considerations we can derive that a test designed through the benchmark configuration file is portable on new test beds. From the flowchart, we can also see that Caliper manages the contract deploying stage although, as in Blockbench, it doesn't explicitly manage the deployment of the SUT. One way to deploy the SUT before the benchmark, is to implicitly encapsulate this task in the "start script", we will notice that the next framework provides a more sophisticated mechanism for SUT deployment.

3.1.4 BCTMark

This framework's aim is to provide a more general and reproducible testing process, in fact it can support almost any blockchain system. Unlike Blockbench, which is mainly an academic tool, this framework offers the possibility

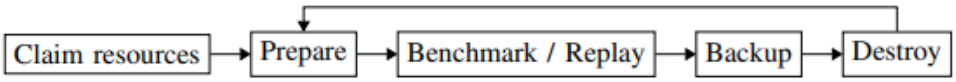

Figure 3.6: Main work-cycle of BCTMark. [19]

to measure system metrics in addition to performance metrics. It also offers a better level of abstraction with respect to the entire testing environment, allowing a better grade of portability. From an operational perspective, HyperLedger Caliper is similar to BCTMark, although the first targets only a finite number of blockchain systems. We are going to show the general workflow of BCTMark and then focus on the two key features that separate this framework from the previous one, that is to say, the possibility to deploy the totality of the components required for the experiments and the network emulation abstraction [19].

First of all, let us show the experimental workflow of this framework in figure 3.6.

Both the "Claim resources" and "Prepare" phases rely on the "deployment configuration file", which describes the required virtual machines upon which deploying the related components. An example of deployment configuration file is provided in Listing 3.1.

```
deployment:
  vagrant:
    backend: virtualbox
    box: generic/debian10
    resources:
    machines:
        - roles: ["dashboard"]
        flavour: tiny
        number: 1
        - roles: ["ethgethclique:bootnode"]
        flavour: tiny
        number: 1
        - roles: ["ethgethclique:peer"]
        flavour: tiny
        number: 2
        - roles: ["bench_worker"]
        flavour: tiny
        number: 1
```

Listing 3.1: An example pf deployment configuration file for BCTMark written in `YAML`[17].

In this case, the Claim Resources phase role is to deploy the required virtual machine "virtual box" with the box "generic/debian10". The Prepare phase handles the deployment of each component under the key "machines". Each component is identified with a role and it is possible to specify the number of instances to deploy.

Another key feature is the "network emulation abstraction", which consists in the possibility of introducing chaotic behaviours between components that interact with each other. This includes, for example, network delays, packet losses and bandwidth limitations, introduced in order to emulate scenarios of real life networks. In particular, users can describe within deployment configuration file these desired conditions.

BCTMark can support practically every blockchain system only if the developer using this framework correctly integrates it. One of the aspects that a developer needs to implement in order to integrate a blockchain system is the *ad-hoc load generation*. This term refers to the BCTMark's necessity of *knowing how to behave* in situations where the behaviour is a blockchain-specific matter. For a developer, implementing the ad-hoc load generation means specifying these blockchain-specific tasks by creating functions that allow BCTMark to send a transaction or to call a contract's method.

Table 3.1: Comparison of the frameworks presented in the state of the art chapter. The table is expanded from BCTMark's paper[19].

	Truffle	Blockbench	HyperLedger Caliper	BCTMark
Targeted Systems	Some blockchains with EVM	Private blockchains	Mainly HyperLedger Systems	Every Blockchain
Deployment Management	Yes (migration script)	No	Yes	Yes
Network Emulation Abstraction	No	No	No	Yes
Portability to new testbeds	Yes (only targeted systems)	N/A (does not manage deployment)	Yes (no management of resources reservation)	Yes (if testbed has SSH)

3.2 Summary

The previously discussed frameworks have been introduced for two purposes. The first is to show the advancement of blockchain and smart contract prototyping frameworks, in respect to the completeness of the prototyping process that is being supported and the number of compatible blockchain systems. The second purpose is to show the different levels of abstraction provided by each framework, as they represent a knowledge base without which we would not be able to address the problems `SCOOP` aims to solve.

To be more thorough, we take from Truffle the completeness of the prototyping process and the contract abstraction concept as well as the orientation towards user-friendliness and smart contracts. Truffle does not address the problems related to blockchain benchmarking and portability, in opposition to the other frameworks. Blockbench is an academic framework that focuses on private blockchain systems and does not assist the user in the smart contract prototyping process in any way, except for the testing phase, which has to be specifically designed for each experiment. Furthermore, Blockbench does not emulate network failures and is not portable to other testbeds since it does not automatically deploy the `SUT`.

HyperLedger Caliper partially addresses these problems, it presents a higher level of portability in terms of test design and targeted systems, but still doesn't cover all the possible blockchain systems. Instead, BCTMark solves the majority of the discussed matters with very few assumptions. Table 3.1 compares the frameworks that have been discussed.

Chapter 4

Design Choices

Understanding the state of the art gave us insight on how `SCOOP` would use, as sub-modules, tools such as Truffle or Caliper. Very different frameworks have been purposely discussed in Chapter 3, in order to highlight in Section 4.1 the problem `SCOOP` aims to solve. The problem consists in having, within the state of the art, a very heterogeneous set of frameworks that present a low degree of discoverability.

Our goal, described in Section 4.2, is to build an entire level that, from one side, can potentially use all the smart contract prototyping frameworks as sub-modules and, from the other, can increase their degree of discoverability towards users, with respect to the principles of software engineering and user-friendliness introduced in Chapter 2. The level, as shown in Figure 4.1, is between the users and all the supported sub-modules.

Studying the tools presented in Chapter 3 (Truffle, Caliper, etc.) is useful in order to build a model that can correctly describe a smart contract prototyping framework at a high level. In particular, the idea of Section 4.3, which is the core of this chapter, is to:

- Define the sub-modules and their role in Section 4.3.1.
- Collect basic facts and assumptions concerning existing frameworks and process each of them with a deductive approach, in Section 4.3.2.
- Let the latter process shape the design of `SCOOP`'s `API` in Section 4.4.1.

The general guideline is to create a level that does not collide with the initial assumptions and that graphically represents the framework's possibilities, state and behaviour in the most intuitive and user-friendly way possible, thus enhancing the degree of discoverability of the frameworks.

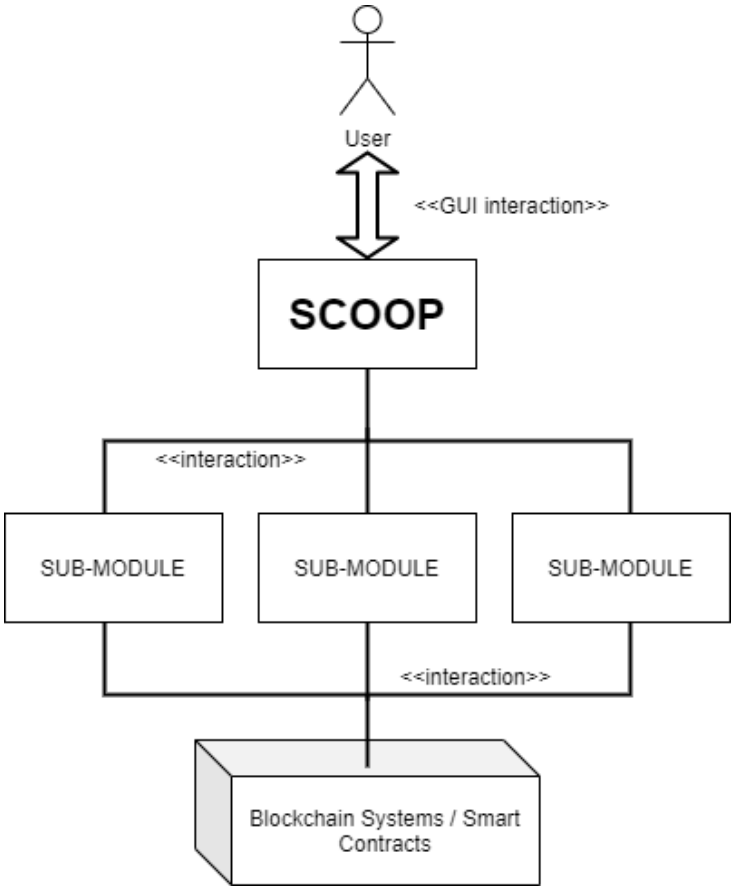

Figure 4.1: SCOOP's position inside the technology stack at a high level.

As we have introduced many new terms and definitions, to help avoiding reader confusion, we have compiled a short dictionary of such terms in Section 4.5 at the end of this chapter.

4.1 Problem

The essential statement that summarizes the content of this section is that the tools presented in Chapter 3 have a low degree of discoverability, especially for unexperienced users.

It is undeniable that the blockchain is a disruptive innovation and, from what has been anticipated in Chapter 2, we can state that smart contracts are too. A common struggle with technology is the one regarding the *discoverability* of products [1], term that indicates how easy it is for a certain category of users to understand which operations a tool supports.

The frameworks presented in Chapter 3 have very different purposes, although they are similar to each other from an operational perspective. In fact, these frameworks implement, partially or completely, the smart contract prototyping `DevOps` cycle. We argue that their low degree of discoverability is due to the following factors, among many others:

1. Users need a `SUT`-specific knowledge base.
2. These frameworks generally need users to provide documents (such as configuration files) in order to work properly.
3. A thorough inspection of the frameworks' documentation is recommended to understand how to use them.
4. The great number of commands and options may cause frustration in the user.

It is important to keep in mind that these considerations have been an important guide for the design of `SCOOP`.

4.2 `SCOOP`'s Goal

`SCOOP`'s goal is to increase the degree of discoverability of frameworks that implement smart contract prototyping processes, mediating between that sort of frameworks and users. In the rest of this section we will properly define the goal by describing the ways in which we can actually increase the frameworks' degree of discoverability.

Let us expand each of the factors listed in Section 4.1:

1. **`SUT`-specific knowledge base**: `SCOOP` makes no assumptions about the underlying technology stack. This is motivated by stating that technology-dependent tasks need to be delegated to what we will later define as the *adapter* of the framework.
2. **Complex configuration steps**: one of the most frustrating aspects of using a software is the setup. One of `SCOOP`'s aims is to make the setup easier for unexperienced users.
3. **Documentation**: the idea is to bring the need of studying the documentation to zero, building a graphical interface with a high degree of discoverability.

4. **Numerosity of commands**: for every piece of technology, certain categories of users tend to use different subsets of the same technology's functions. Our solution therefore gives the adapter's developer carte blanche on choosing which functions and operations of the same framework he will project to the user.

In other words, SCOOP's goal is to introduce a new level between users and smart contract prototyping frameworks, building an high level graphical interface that is generic towards those frameworks' purposes and, in general, towards the technology stack underneath. At the same time, our graphical interface must increase those frameworks' discoverability, by addressing the factors 1-4.

4.3 SCOOP

The first point to address is the one regarding the mechanism that will allow us to attach a framework to SCOOP, namely the *adapter*, properly defining, in Section 4.3.1, what exactly a SCOOP sub-module is.

Having stated the goal of SCOOP in Section 4.2, the problem is narrowed down to finding those assumptions, parameters, and characteristics that can correctly describe any framework that handles the prototyping cycle of smart contracts, as Section 4.3.2 reports.

Once those observations have been made, it is possible to summarize them and reflect on the implications they introduce in order to design the SCOOP's API in Section 4.3.3.

4.3.1 Sub-Module: Framework & Adapter

Making assumptions about common features among frameworks is needed in order to build the SCOOP API those frameworks will be attached to. In this context, any given instance of this API is named *adapter*, which is the component that makes the modularization possible.

As shown in Figure 4.2, a particular case is the one in which multiple adapters use the same framework. For example, a developer could choose to expose only a subset of the framework's commands, another developer may build a new adapter, improving the performances of the already existing one, as well as exposing a different set of the underlying framework's functions.

From these considerations, we can derive that the key component of a sub-module is its adapter.

Let us bring the focus on finding the frameworks' assumptions that will allow us to build the API

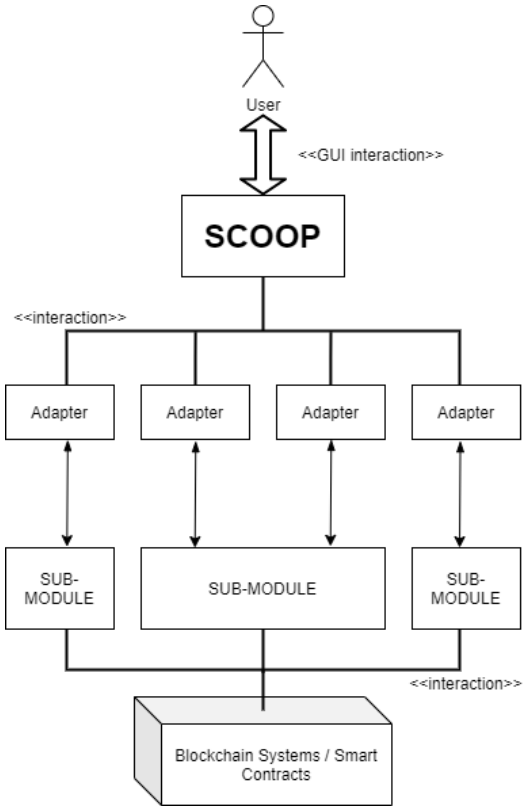

Figure 4.2: SCOOP's position inside the technology stack, considering the role of the adapter.

4.3.2 Frameworks' Assumptions

This section will provide a list of common features observed among frameworks that manage smart contract prototyping processes. We can distinguish those features in two categories, the ones concerning a framework in general and the ones regarding the smart contracts prototyping process in particular. We will discuss them in this order.

Generally, it is reasonable to assume that frameworks need the user to provide documents, such as configuration files, in order to function properly. In addition, SCOOP's graphical interface must be user-friendly, so it is necessary to find a standard procedure to acquire, edit, and delete documents in such a way that the user would not need to edit them manually.

Another consideration is that frameworks have commands, to which are associated parameters and behaviours, with the latter generally expressible using function calls that may produce results. In turn, a computer program, and therefore a framework, owns an execution environment in which com-

mands are executed.

A more abstract assumption concerns the workflow: any prototyping framework has a main sequence of events, thus it would be useful to find a structure that correctly describes the frameworks' workflow, for many reasons. One of the reasons is that it would be easier to organize the graphical workspace of `SCOOP` if we had a separation of concern between different phases, steps or windows within the application.

The far more specific, though trivial, assumption we can make about these framewoks is that, by definition, they test smart contracts. To be more clear, let us think about the example "Simple Storage" in Section 2.2.2. A user that aims to test deployed versions of this smart contract is interested in selecting the desired storage among all of the deployed instances of "Simple Storage" and, once he has done that, he may want to activate some of the deployed contract's functionalities, setting or retrieving a value from the storage.

Another observation is that a smart contract is adaptable to a number of blockchain systems, the results would be several contracts with different syntax and identical semantic. At this point of our analysis, it seems reasonable to assume that, at some level of abstraction, it is possible to describe the interaction with a smart contract regardless of the language it is written in or blockchain platform it is deployed to.

The following list summarizes the assumptions collected in this section:

1. Frameworks need users to provide documents.
2. Frameworks have commands and need to present their results.
3. Framework's commands have to be executed in the frameworks' environment.
4. Existence of a significant structure for the workflow, among different frameworks.
5. Existence of an abstraction for smart contracts, among different blockchain systems andor languages.

In Section 4.3.3 these assumptions are deepened in order to design `SCOOP API`.

4.3.3 Building `SCOOP`'s `API`

For the rest of this section, we analyze the different ways in which we can address the basic assumptions listed in Section 4.3.2. The main procedure, for each assumption, will be to ask questions about the nature of the assumption,

focusing on edge cases in order to cover the greatest number of scenarios as possible and make important observations.

The answers to those questions will provide insight on how to finally build the `API` developers will implement in order to attach a framework to `SCOOP`.

The questions asked to develop `SCOOP` are the following, with respect to the points listed in the previous section:

Q1 Which kind of documents frameworks need users to provide? Is it possible to find an abstraction to describe their common properties?

Q2 How does a framework expose its commands? How are they made available to the user? How will the user visualize the corresponding results?

Q3 In which environment are the framework's commands executed?

Q4 How do we model the workflow of a generic framework?

Q5 How will the user interact with smart contracts? Is it possible to find a smart contract abstraction?

`Q1`: Artifact Abstraction

Which kind of documents frameworks need users to provide? Is it possible to find an abstraction to describe their common properties?

Our first step towards a document abstraction is to give it a name, we therefore call *artifact* the class describing all the properties that any kind of document within the `SCOOP`'s workspace must have.

In order to design the artifact class, we must reflect on some properties that may characterize the totality - or a reasonably large category - of documents frameworks use. We start by asserting that documents may be required in order for the sub-module to work properly, we therefore extend the artifact class with the property "required".

The sub-module's developer may want to provide the user the possibility of adding, editing and deleting documents to the workspace, with the important note that we want to avoid forcing the user to manually edit files. Intuitively, avoiding raw text editing on behalf of the user is possible only under certain circumstances.

We find that documents such as configuration files present a simple structure that can be mapped to a form when it comes to represent them graphically. In this case, the raw editing of the artifact would be avoided, since editing a form is much simpler.

An example may be the Truffle configuration file shown in Listing 4.1, transformed into the two forms[1] displayed in Figures 4.3 and 4.4.

```
module.exports = {
  networks: {
    my_network: {
      host: "127.0.0.1",
      port: 8545,
      network_id: "*" //"*" Matches wth any network ID
    }
  },
  compilers: {
    solc: {
      version: "^0.8.0"
    }
  }
};
```

Listing 4.1: An example of Truffle's configuration file.

Documents with such a simple structure are describable as a key-value data set, for example the "host" key in Listing 4.1 is associated to the value "127.0.0.1". Intuitively, the "host" key can be associated to a wide range, though limited, of values, in turn other keys may even accept every kind of string. In order to describe in general what kind of data documents allow within their syntax, we introduce the concept of *schema*.

A schema describes textually what kind of values can be associated to a given key. We find in Section 5.1 that schemas have a one to one relationship with forms, in the sense that there is one and only one form corresponding to a given schema and viceversa. Being able to associate a schema to a set of documents with the same syntax is a very important feature, since we can increase `SCOOP GUI`'s degree of discoverability by showing the user which values he can insert in a given field, or even which fields are required for the form to be valid.

It is too strong to assume that documents' syntax, in general or for the most part, can be perfectly described using schemas. Instead, it is reasonable to assume that a lot of documents' syntax are *similar* to what we can represent with a schema. For example, Truffle's configuration file in List-

[1] As anticipated in Section 2.1.1, these two forms are one of the many models that correctly describe Truffle's configuration file. A developer who is building another adapter may choose to prompt to the user all of this information using just one form. This is a design choice and it is expressly left to the developer.

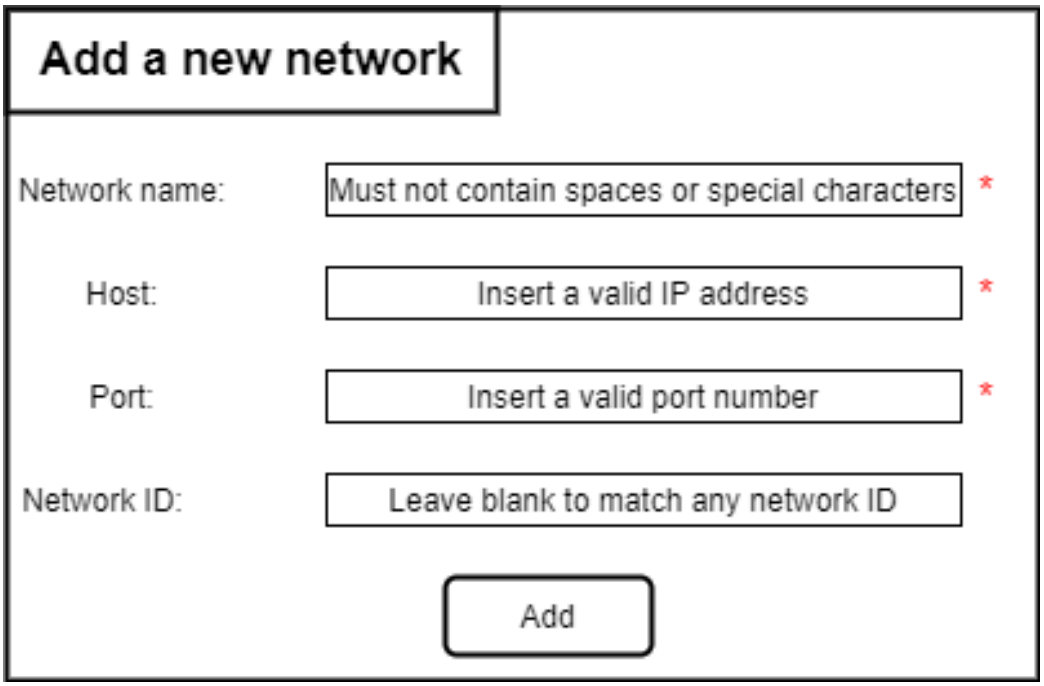

Figure 4.3: First part of the configuration file.

ing 4.1 does not perfectly map with the simple key-value data set produced as output by forms in Figures 4.3 and 4.4. In fact, this configuration file presents a different syntax and holds a greater amount of framework-specific information, such as the "module.exports" line.

SCOOP makes no assumptions about the way sub-modules map the form results to the correct documents' syntax, on the contrary, the majority of the frameworks will need to perform some sort of transformation of the form results in order to map those result to a document. Our solution is to extend the artifact class with a *parser*, defined as the piece of code that, having fixed a schema and the associated form, transforms (parses) the form's result into a document with the correct syntax.

In addition to the transformed input, the parser also returns other kinds of information, such as the path and filename that SCOOP uses to save the transformed form result, which will be the actual document that the framework will manipulate.

Even when documents are perfectly representable using forms, two observations need to be made.

Observation (1): the document's syntax may allow nested structures. In terms of key-value associations, the value may in turn be a key-value object. This can be managed in two different ways:

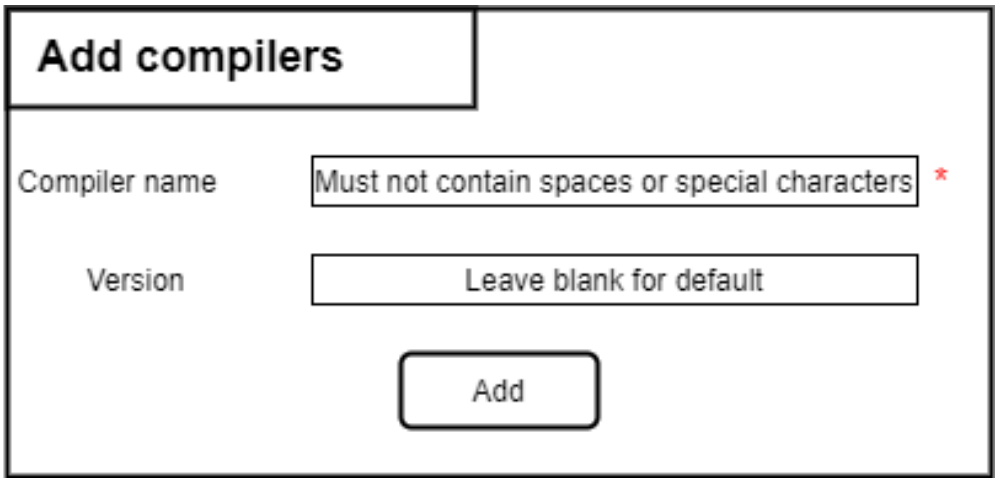

Figure 4.4: Second part of the configuration file.

- Explicitly, by displaying the nested structure within the same form, as shown in Figure 4.5.
- Implicitly, by displaying different layers of the nested structure in just as many different forms, as the previous Figures 4.3 and 4.4 display[2].

In Chapter 5, the mechanism used to represent forms will not support nested structures, so it will be an adapter developer's responsibility to represent the nested structure implicitly and in the most user-friendly way possible.

Observation (2): concerns the multiplicity of values associated with a given key. For instance, in the Truffle configuration file in Listing 4.1, users have the possibility of associating several network objects to the "networks" key. In this case, since nested forms will not be supported, our suggestion is to associate to the critical entity (Network) a separated form, prompting the latter to the user as many times as he needs and then parsing the produced result by adding it within the document that encapsulates the entity (Truffle configuration file).

Alternatively, the document's syntax and semantic may require some objects/entities to be unique, such as the compiler version in Listing 4.1, about which the user can indicate just one preference. We therefore extend our

[2]This option can be a good trade-off between exposing the entire, nested, and monolithic file structure in the same form, versus completely removing the nested structure, which cannot be a good solution since it presents a low degree of discoverability (of the document's structure) towards the user.

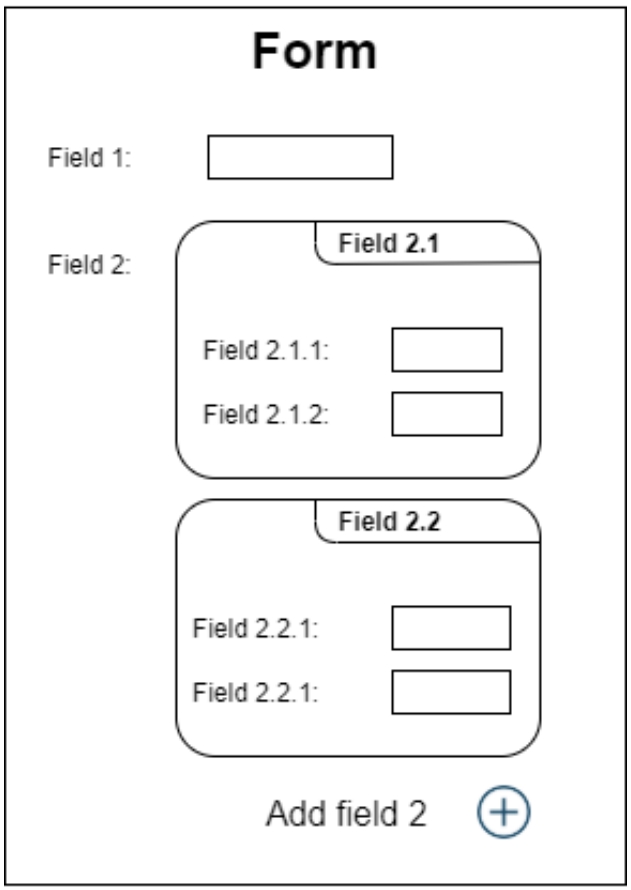

Figure 4.5: A form with a nested structure.

artifact class with another indicator, the "iterable" flag, that tells SCOOP whether a given artifact instance can produce multiple documents or not.

The user might want to edit documents produced in output by a parser of a certain artifact instance. The problem is that we have not yet designed a mechanism to invert the process performed by the parser. Inverting the parser's process means generating the key-value data set starting from the raw document syntax, in order to use that key-value data set to initialize the form associated with the artifact instance in question, as shown in Figure 4.6.

A crucial observation to solve the problem of inverting the parser method is that an iterable artifact instance is in a one to many relationship with documents produced in output by its "parser" method, SCOOP therefore needs to distinguish one document from the other. Our solution is to assign an identifier to each document given in output by the parser. SCOOP therefore

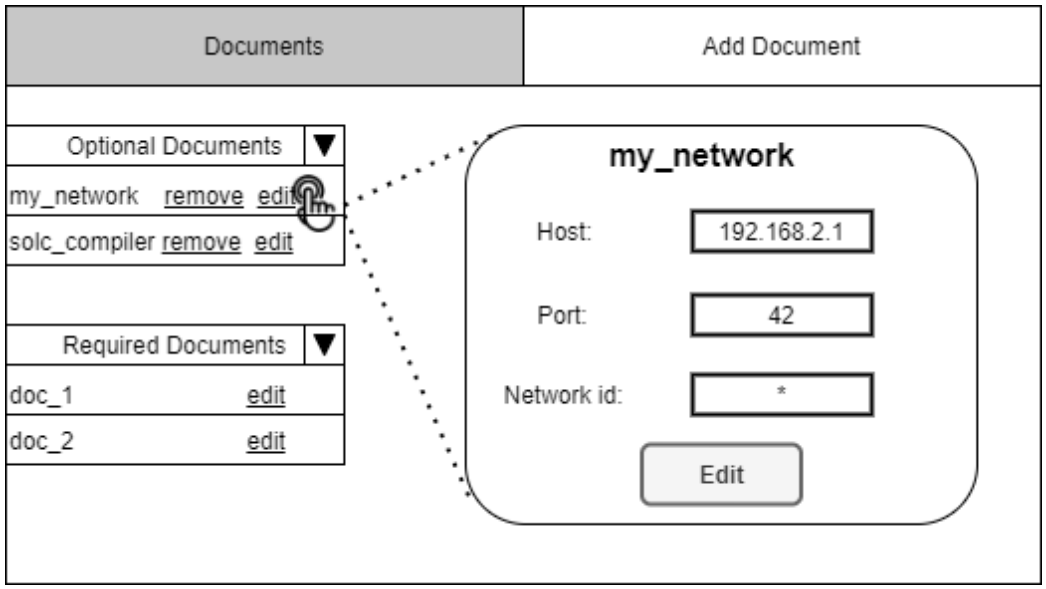

Figure 4.6: The initialization of the form associated to the artifact the user wants to edit.

needs to maintain a database that associates each artifact instance to a set of document identifiers.

We call *reviver* the inverse function of the parser. Wrapping up the previous considerations, the reviver of an artifact instance takes as input a document identifier and returns the key-value data structure that will be used to initialize the form associated with the artifact instance's schema, which is the data that the user originally submitted using the same form.

To summarize the contents of this Section, Figure 4.7 shows a class diagram of an artifact.

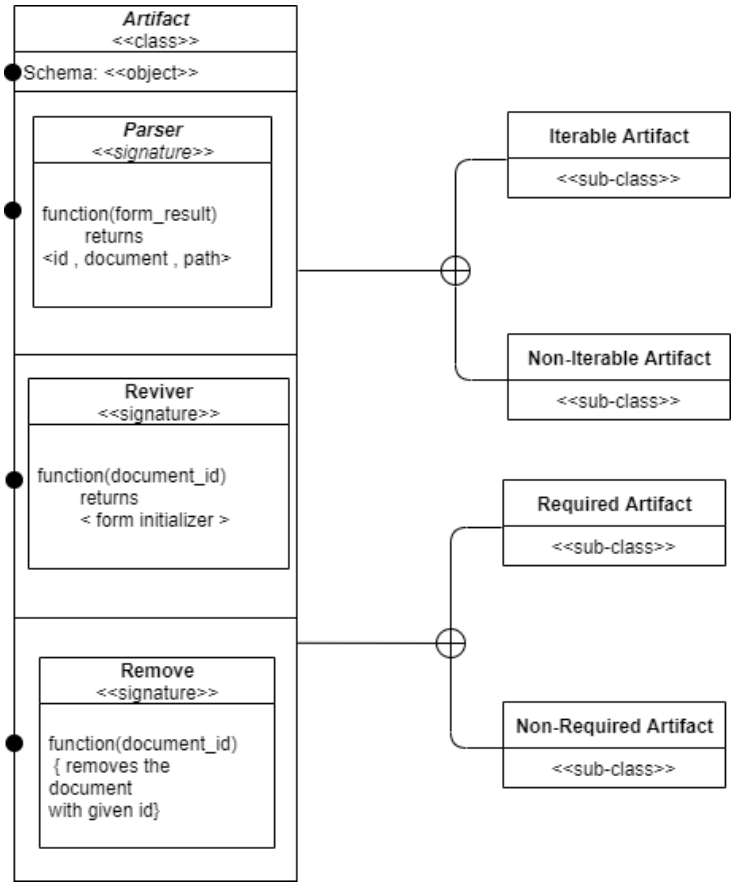

Figure 4.7: Properties of an artifact within `SCOOP`'s `API`.

Q2: Command and Functionality

How does a framework expose its commands? How are they made available to the user? How will the user visualize the corresponding results?

For the rest of this chapter, the methods that the adapter exposes through the API will be referred to as "functionalities", whereas the framework's native methods will be called "commands".

The adapter's developer needs to expose commands through the API, the result is a functionality. Those functionalities may or may not correspond to the framework's commands. When functionalities correspond exactly to the commands, the functionalities are simply *stub* methods, in the sense that their behaviour is limited to invoking the commands with the parameters inserted by the user using SCOOP's GUI.

In the second case, the adapter's functionalities do not map with the framework's commands. This could be useful in all of the cases where the adapter's aim is to:

- partially evaluate a certain command's parameters.
- expose through the API a subset of the commands.
- include additional functionalities.

SCOOP's API will support each of these scenarios.

Let us focus on the mechanism the developer uses to expose functionalities implementing SCOOP's API.

Our solution is to extend the API with a list of objects corresponding to functionalities. In this case, each functionality is an object with the properties "name" and "parameters". From a graphical perspective, the user activates a functionality of the adapter by:

- selecting a command from the ones exposed in the API.
- inserting all of its parameters using a form.
- ordering the execution of the command.

In turn, SCOOP invokes the adapter to actually execute the functionality, which finally may or may not run the framework's command(s).

SCOOP does not add constraints about the output of the functionalities. In order to make the graphical workspace transparent towards the sub-module's activities, the API will provide a textual monitor to which the sub-modules will send updates concerning the functionalities' results. The textual monitor is necessary, since not every functionality's return value can be "pretty-printed". This is the reason why log files are generally used when the results

of certain operations cannot be condensed in a few rows. Therefore the textual monitor could be a log file manually inspected by the user via a file manager, or it may be a dedicated window within the SCOOP's GUI. We choose the second option. The monitor may also be used for general updates about the SUT's state.

We present three optimization that will make this mechanism more generic and user-friendly.

Optimization (1): given a method, its parameters may accept only a finite set of values. So the API supports, for each parameter, an additional field that indicates the set of values the parameter accepts. This optimization is not necessary, since the user who tries to insert an invalid value for a parameter will eventually receive a negative response from the adapter. Nevertheless, in a user-friendly approach, it is important to let the user be aware of which actions he can perform using the application, in order to prevent frustration.

Optimization (2): we notice that the set of allowed values for a given parameter could vary in time. An example of this fact: the method "compile" accepts "contract name" as parameter, which generally corresponds to the name of the file containing the source code of the homonymous contract. If we statically assign to "contract name" the list of available source codes in the file system through the adapter, the user will not be able to add a new contract's source code and see the related entry when inserting the parameter. A solution to this problem may be to retrieve the values at runtime via a method called "getValues()". In this case, the adapter developer associates to "contract name" a function that returns a list of values rather than returning the list as a constant. In our example, this function may scan the file system and retrieve the list of contracts' source codes stored in a given directory.

Optimization (3): the textual monitor can be limiting, since the user might be interested in knowing a subset of the functionality's reply. For example, when the functionality's objective is to compile a set of source codes, the user is interested in knowing if the compilation is successful or not. The user is generally not interested in a detailed log of the compilation, and therefore in consulting the textual monitor, except the case in which the latter process has been aborted for some reason. A solution to this problem is to implement a publish-subscribe system, through which the user subscribes to one or more *outcomes* the functionality publishes. In practice, the user will select a functionality, insert its parameters and choose which outcomes of the functionality he is interested in receiving. The functionality class must therefore be extended with a set of outcomes the user may decide subscribing to.

Figure 4.8 shows the properties of a functionality, according to the model

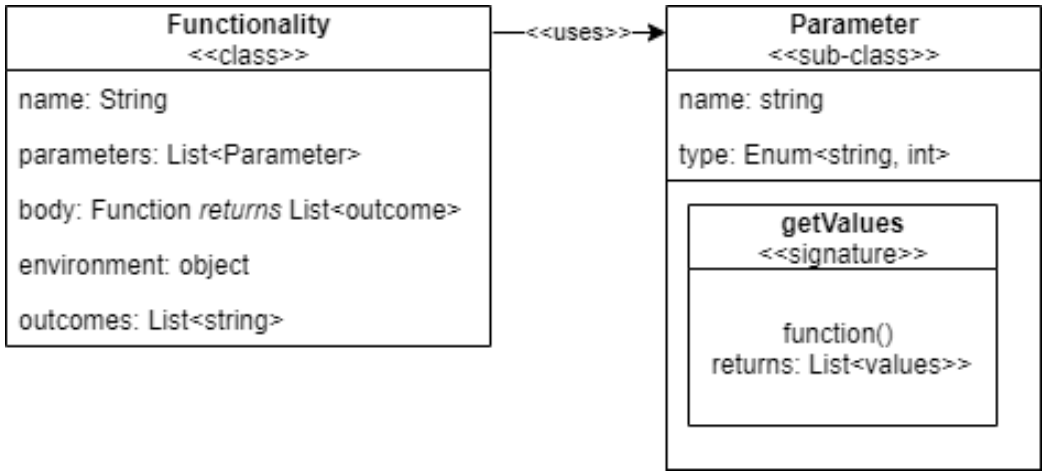

Figure 4.8: Class diagram of a functionality within `SCOOP`'s `API`.

just presented.

`Q3`: Environment

In which environment are the framework's commands executed?

The previous section does not address an important issue, that is to say, "where" to execute functionalities, in terms of environment. In fact, the adapter's functionalities will generally have an environment within which they'll need to be executed. The only case in which the environment is not a discriminating factor is the one where the functionality's behaviour is invariant respect to the parameters and, in general, doesn't rely on external resources such as the file system.

When an environment is needed, the solution could be to extend the `API` with an object that represents the execution environment of the sub-module, or even to add the possibility to assign a different environment to each of the functionalities, with the latter being at a higher granularity level. `SCOOP`'s `API` supports both options, using the global environment when the specific functionality's environment is not provided.

`Q4`: Workflow

How do we model the workflow of a generic framework?

A typical workflow of a framework includes its configuration and usage. In this setting, we refer to configuration as the general task of including documents within `SCOOP`'s workspace. Instead, the usage concerns the execution of functionalities by the user, and it is considered the heart of the testing

process. Since a lot of tasks are generally repeated by the same user, this step will allow the latter to create a "test script", which is an ordered sequence of functionalities, the ones exposed through the SCOOP's API by the adapter.

An important feature of tests in general is repeatability, thus this test scripts are memorized in order to allow the user to execute them at a later time. The central idea is that the user should be able to customize and memorize common sequences of tasks by building several test scripts and executing them in any given moment. An instruction of the test script maps out with a functionality of the adapter, and the user inserts the parameters of that specific functionality through forms. We can now split the usage phase in two different ones, the test script preparation and the test script execution. Of course, the user must be able to edit the test scripts he built in a user-friendly way.

Building SCOOP's GUI, at first the idea of graphically waterfalling these three phases in subsequent steps seemed reasonable. This solution is too simplistic, since users often tend to reconfigure the framework while they use it, according to their needs. An improvement consists in dividing configuration and usage in parallel tabs, so that the user is able to configure or test the sub-module anytime.

The nature of artifact instances exposed by the adapter shapes users' perception of the workflow, we pinpointed in Section 4.3.3 two classifications for artifacts: iterable/non-iterable and required/not-required. In fact, SCOOP's GUI must allow the user to add documents one or multiple times, according to the iterable flag of the pertaining artifact instance. The GUI also must force the user to insert add a given document if the required flag of the associated artifact instance is set.

As we already observed, when using a framework, users tend to repeat tasks. For example, in Truffle, whenever the user updates the source code of a smart contract, he may need to recompile the entire project. The user could do it manually or choose to automatize the task by using features like "file system watchers", this feature allows the user to execute a script anytime a sub-portion of the file system changes. This is a desirable feature to include in the API. A similar optimization may be the one Caliper adopts, and it is to associate the execution of scripts to different parts of the workflow. As explained in Section 3.1.3, Caliper's API allows the developer to add a "start script" and an "end script", to be executed respectively before and after the Caliper Manager process does its work.

In SCOOP's API, these two techniques from Truffle and Caliper are mixed. We add to the "start script" and "end script" three additional scripts, the "update script", the "pre-test" script and the "post-test" script. The "update script" must be executed every time an artifact is added, edited or removed,

similarly to what happens with the "file system watch" feature. The "pre-test" and "post-test" are executed respectively before and after the execution of one of the test scripts built by the user. This is due to the fact that engaging in the test script execution phase may involve tasks such as starting servers, proxies, third party tools, and so on. Thus, these two pre-test and post-test scripts will help the adapter developer run and kill programs that come in support of the prototyping process.

`Q5`: Smart Contract Abstraction

How will the user interact with smart contracts? Is it possible to find a smart contract abstraction?

Up until this section, our `API` does not include any feature that is specifically related to smart contracts, in fact the assumptions discussed in the previous sections are valid for the majority of the frameworks.

As reported in Section 3.1.3, every data in support of the prototyping process is defined as an artifact. In this setting, smart contracts' source codes inserted by the user are considered artifacts too, although smart contracts, as entities that have a behaviour, are not. The point at which smart contracts start to behave differently than artifacts is when the user is interested in interacting with them.

In particular, a characteristic of smart contract prototyping frameworks is that commands may refer a smart contract. As anticipated in Section 3.1.1, Truffle offers the "truffle-contract" library, which allows the users to interact with smart contracts without using the far more complex "web3" library. The "truffle-contract" library allows to reference documents such as smart contracts' source code and, given a smart contract, it can also distinguish among its different deployed instances, allowing the user to call their methods. In other words, Truffle provides a *smart contract abstraction*.

This fact leads us to the consideration that the sub-module's functionalities may contain references to smart contracts too, in the sense that one or more of the parameters could refer some properties of a smart contract. We individuate three possible ways in which a smart contract can be referred to within a functionality's parameter:

1. The parameter refers to the smart contract's source code or entity. (class)

2. The parameter points to a specific deployed version of the smart contract entity. (instance).

3. The parameter accepts a smart contract method call, in the sense that the actual parameter will encapsulate information about the smart contract's deployed instance, the method the user needs to call and the parameters of the method.

The API needs a mechanism to retrieve the properties of smart contracts that live in the workspace. As reported in Section 2.2.2, the ABI provides a unique description of a smart contract entity which is language invariant. In addition, different blockchain virtual machines support a different ABI formats, so this mechanism must be built accordingly.

Our solution is to extend the SCOOP's API with the interface "VM Adapter" that, once implemented, allows to retrieve and transform (parse) the ABIs into a more maneuverable structure. It is worth pointing out that very few "VM Adapters" need to be implemented, since very few blockchain virtual machines exist. The "VM Adapter" transforms the ABIs in a more manageable structure for SCOOP, namely the "Contract Descriptors". The "Contract Descriptors" are generated by a VM-specific method in our API, the "parseABIs". This structure concerns all of the contracts that have an ABI and contains metadata such as:

- The list of contract entities (classes).
- The list of deployed contracts (instances).
- The list of the contract's methods for each contract
- The list of the contract method's parameters.

Since the user is able to add, delete or remove artifact instances associated with smart contract entities, the "Contract Descriptors" structure must reflect those changes. This means that the structure has to be updated every time a smart contract artifact is added to the workspace.

In order to adapt to the majority of the sub-modules, SCOOP should not make assumptions about where the sub-modules store the contracts' information, although the majority of the frameworks store the smart contracts' ABIs in a dedicated path, generally called "build path". In addition, our framework should not make assumptions about the format in which those ABIs are stored, although the JSON format is widely adopted by most of the known tools and any other format can be easily translated into JSON. Our solution to this problem is to extend the API with an optional method named "getABIs" that, once invoked, provides the ABI list in a JSON format.

Developer's frustration needs to be avoided as much as the user's. We just pointed out that most of the frameworks store the contracts' ABIs in the

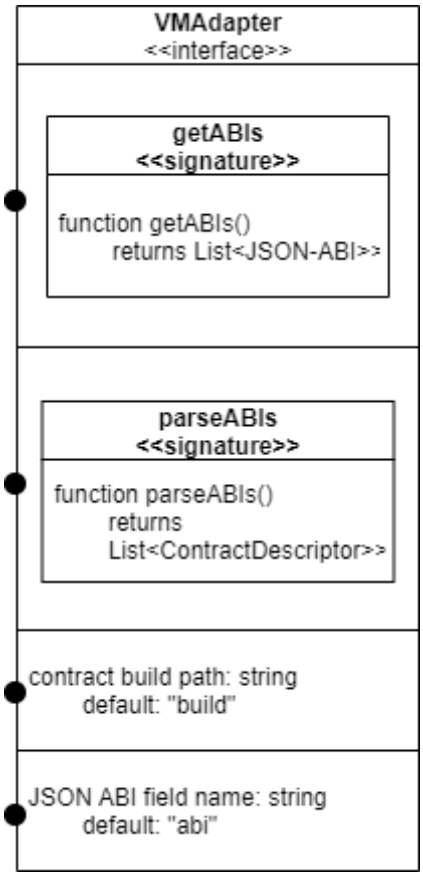

Figure 4.9: Class diagram of the `VM` Adapter within `SCOOP`'s `API`.

file system and in a specific path, thus it is important to introduce a default behaviour that handles the latter case, so that the developer does not need to write more code than necessary. Our solution is to extend the `API` with two optional properties:

- The contracts' "build path", within which we find the `JSON` descriptors of the contracts.
- The "descriptor's field name" that contains the `ABI` object.

When the adapter does not implement the "getABIs" method, `SCOOP` will check if these two properties are defined. If one of the two properties is not defined, a default value will be assigned. Finally, `SCOOP` looks for contracts' `ABI`s within a default build path and a descriptor's field name.

The class diagram of the `VM` Adapter is displayed in Figure 4.9, with respect to the above considerations.

4.4 The SCOOP's Architecture

In this section we pinpoint the crucial aspects of SCOOP's architecture. This includes both static and dynamic aspects of the architecture. Firstly, we are going to circle back to the SCOOP's API we started building in Section 4.3.3 by providing a static component view of the latter in Section 4.4.1. Secondly, a global bird's eye view of the entire stack in question is analyzed in Section 4.4.2. Finally, we provide several behavioural views of the main user stories, by highlighting the role of the other components in the system.

4.4.1 SCOOP's API

In Section 4.3.3 we processed the assumptions, issues and constraints presented in Section 4.3.2. This process led us to considerations that helped creating the building blocks of the API. Let us briefly summarize the content of those sections in order to provide a clearer, global view of SCOOP's API.

Through the API, the Adapter is able to:

- Expose schemas that model any type of data that can be described as a key-value structure. The schema is in a one to one link with a form, and the latter is prompted to the user, who will fill it with concrete data. Submitting a form produces an object that is interpreted and stored as a document through the parser. The user edits data that has already been submitted using a form, this time initialized with the same data *he originally submitted. This initialization is possible thanks to the reviver, which is the inverse function of the parser and therefore converts the document into the object that is finally used to initialize the form. Schema, parser and reviver, together with the iterable and required flags, represent the properties of the artifact class or abstraction. We therefore name an instance of artifact class "artifact instance". The iterable flag tells SCOOP whether or not the artifact instance can have more than one document associated, whereas the required flag is used to force the user creating at least one document that maps to the related artifact.

- Expose functionalities that may or may not map with the underlying framework's commands. Functionalities have a name that is used to refer them and may need an environment in which their body needs to be executed. A functionality may accept parameters, these parameters have a name and can be primitive or non-primitive, non-primitive parameters are the ones related with the interaction with smart contracts and that require a particular attention in the rendering task. Since

some parameters may accept a limited set of values as their domain of existence, the parameter class also owns a "getValues" method that allows to retrieve the allowed values at runtime. When the execution of a functionality terminates, the return value is an object whose keys are listed in the outcomes field. The user subscribes to one or more outcomes before saving the script, only the outcomes he chose will be displayed after the execution of the script.

- Use the global environment object as default, which is another component of the `API` that the adapter provides, when an environment is not provided for a specific functionality.

- The workflow's design choices involve both front-end and back-end. The front-end is structured in parallel tabs so that the user can move his attention from one to the other as easily as possible. The tabs allow to add/open/edit/remove documents, create/save a test script and choose which script to run among the saved ones. The workflow aspect of the back-end consists in allowing the adapter's developer to automate tasks using scripts that the adapter associates to different phases of the `SCOOP`'s life-cycle, as well as to actions performed by the user. In particular, different scripts are associated to the start/end of the sub-module's lifecycle and to actions performed by the user, such as updating an artifact and saving or running a script.

- Smart contracts are considered documents and can be easily associated to a a schema since it is possible to open, edit and remove them exactly as it happens with other kinds of documents. From the moment at which the smart contract is deployed, we need other mechanisms that allow the user and `SCOOP` to interact with it. This implies being able to visualize the smart contract's methods via the structure "ContractDescriptors", task that reduces itself to retrieving the `ABIs` of the contracts in whatever place the sub-module has stored them and interpreting them. There is a standard that `SCOOP` can adopt in order to relieve the adapter developer the task of implementing these behaviours. Since retrieving the `ABIs` implies knowing where the latter are stored and interpreting them requires knowledge about the blockchain `VM`, the standard may not be valid. We therefore extend the `API` with two methods, "getABIs" and "parseABIs" that respectively return the list of `ABIs` in a familiar `JSON` format and the "ContractDescriptors" structure.

- We extend the `API` with the Log Monitor, a mechanism that allows the

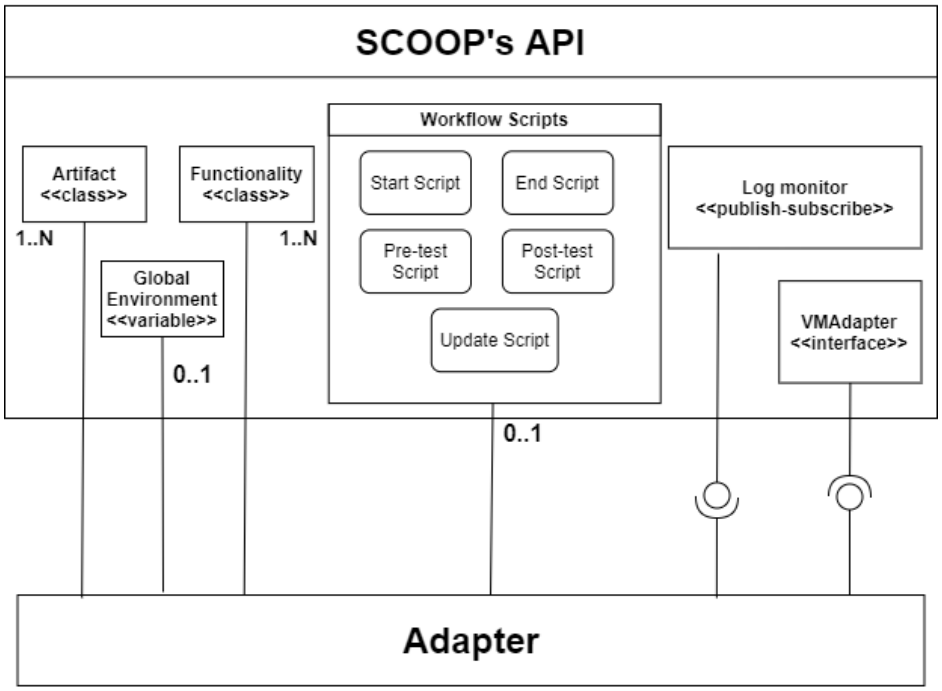

Figure 4.10: Summary of the structure of `SCOOP`'s `API`.

sub-module to send asynchronously notifications about the `SUT`'s state.

Figure 4.10 displays the result of this summary.

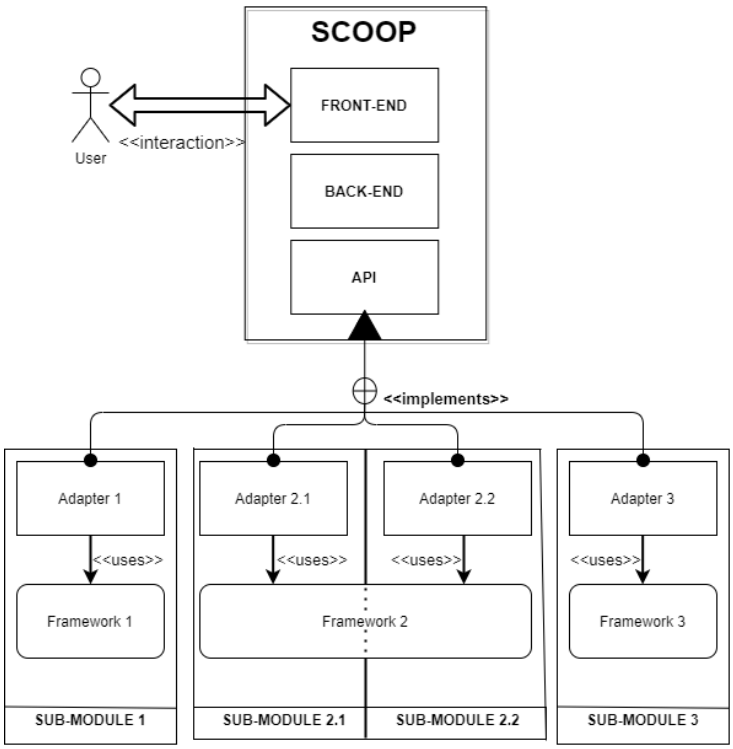

Figure 4.11: Bird's eye view of `SCOOP`.

4.4.2 Bird's Eye View

As displayed in Figure 4.11, `SCOOP` is placed between the user and the sub-modules. The user interacts via the front-end, whose main tasks involve rendering forms and updating the tabs basing on the actions the same user performs. Actions performed by the user are interpreted by the back-end and generally involve the sub-module, that in turn interacts with the back-end through the adapter. The adapter implements the `API` in order to expose the underlying framework's commands, multiple adapters may be provided for different frameworks. The sub-module is abstractly defined as the union of adapter and framework. In fact, when the application starts, the first choice the user makes concerns which sub-module to use.

4.4.3 Dynamic Views

In the first part of this section, we briefly analyze the general lifecycle of `SCOOP` and we deepen this aspect by analyzing several user stories in form of dynamic views.

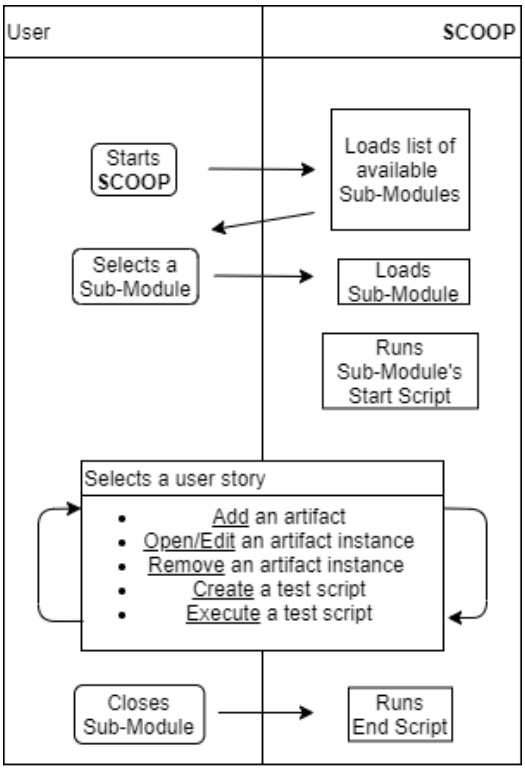

Figure 4.12: Sequential diagram of the SCOOP's lifecycle, at a high level.

Lifecycle

When a user starts the SCOOP application, the first task that the application performs is to load from an external source all the available sub-modules. The user chooses which module he wants to use and SCOOP subsequently loads the sub-module into its environment, this implies loading the adapter as well as the framework. Once the adapter is loaded, SCOOP also runs, if present, the start script the adapter exposes.

From that moment on, the user may perform different tasks, each of which being thoroughly expanded in the following paragraph. After performing operations that concern adding, opening, editing or removing an artifact, SCOOP's back-end executes the update script. The execution of a test script instead takes place after (resp. before) SCOOP's back-end runs the pre-test (resp. post-test) script, if it is provided. In addition, before closing the application, the end script is executed as well. All these steps are summarized in Figure 4.12.

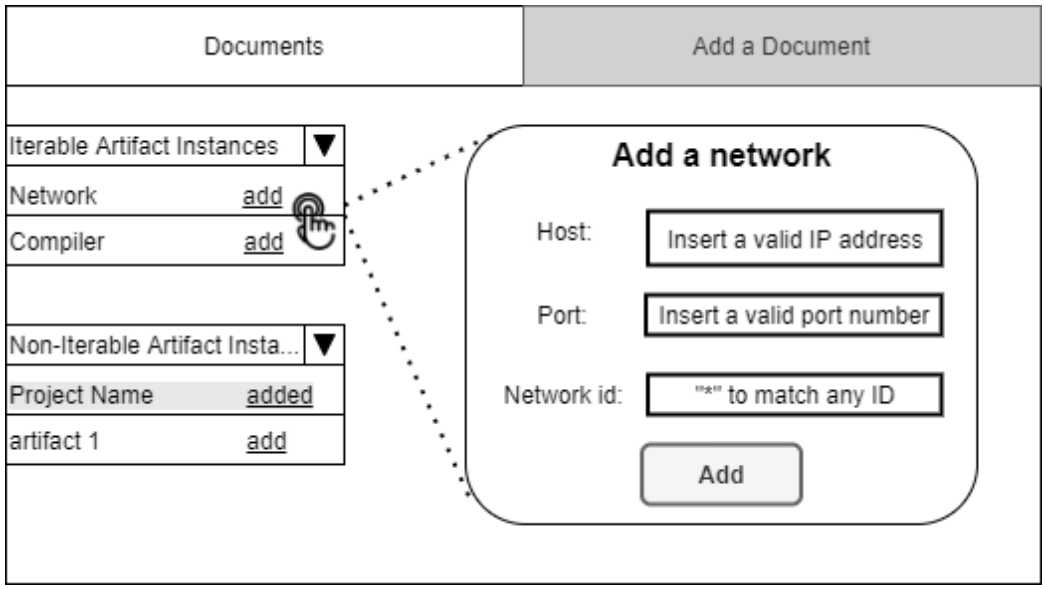

Figure 4.13: Graphical view of how the user may add artifacts to the workspace.

User Stories

In this section, we are going to examine in depth the main tasks that the user can perform, highlighting the role that SCOOP, the adapter and the framework have in the tasks' execution.

Add a Document In order to add a document, users select an artifact instance name from the available ones as shown in Figure 4.13.

After the user selects an artifact instance, SCOOP uses the sub-module's adapter to retrieve the associated schema and uses that schema to render the form accordingly. The user compiles the form and submits the result, as Figure 4.14 explains. The form's result is a simple object that SCOOP's back-end passes to the artifact instance's parser as shown in Figure 4.15. The return value of the parser contains the parsed result that represents the document, the path in which to store the latter and the document's generated identifier.

Open/Edit a Document In order for the user to be able to edit the document, the latter's name is displayed in the workspace within a list similar to the one displayed in Figure 4.6. Once the user identifies the document he wants to edit, *he should be able to modify the form he had previously submitted. To do so, the adapter uses the reviver of the artifact instance to

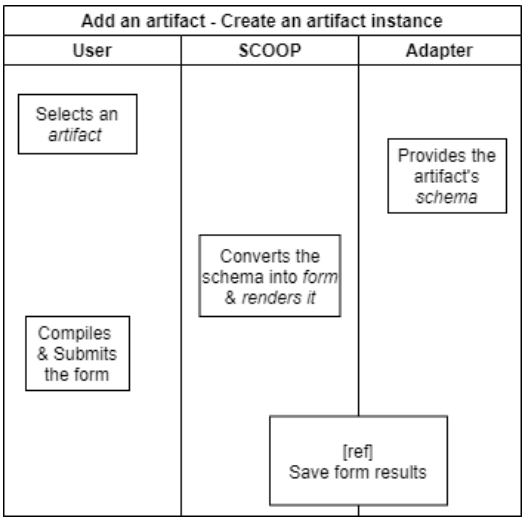

Figure 4.14: Tasks that the user, SCOOP and the Adapter perform in order to add an artifact

transform the content of the specific document (which has an identifier) into an object that can be used to initialize the form. SCOOP renders the form and initializes it with the object in question. The user may now edit the form or decide to just visualize its content, and from that moment on the task is the same as the one we have modeled in Figure 4.15. Figure 4.16 summarizes this section.

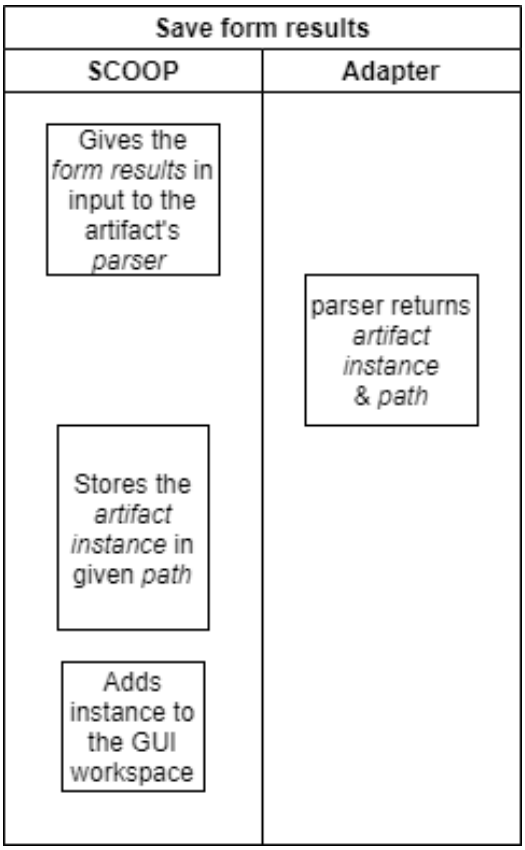

Figure 4.15: After the user compiles and submits a form, what steps does SCOOP and the Adapter perform in order to add elaborate and store the form results.

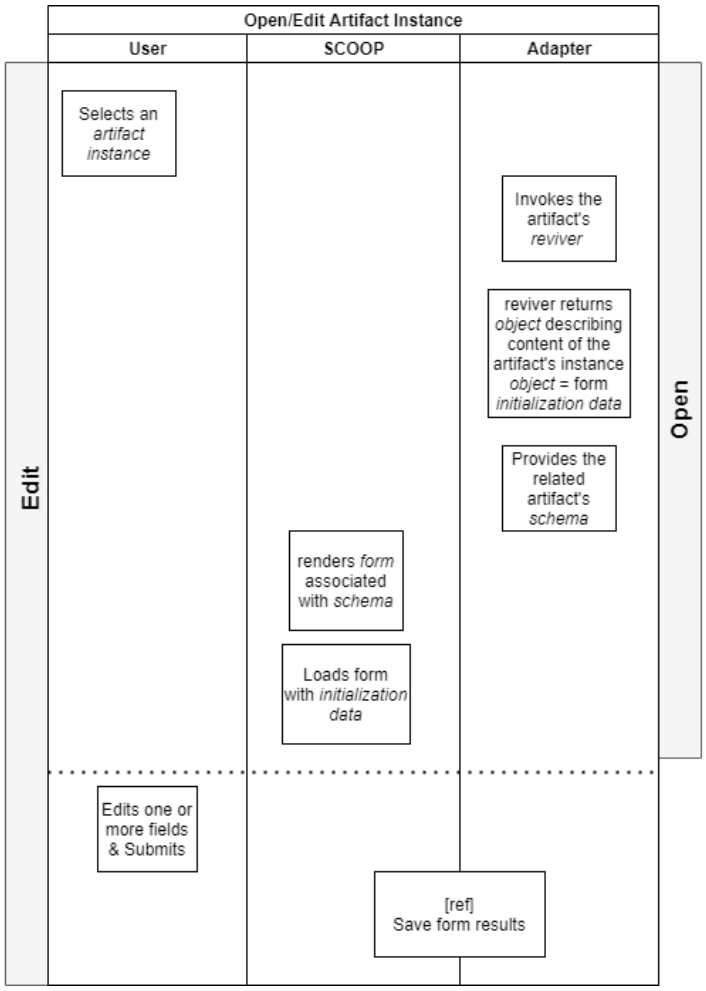

Figure 4.16: Tasks that the user, SCOOP and the Adapter perform in order to open or edit a document.

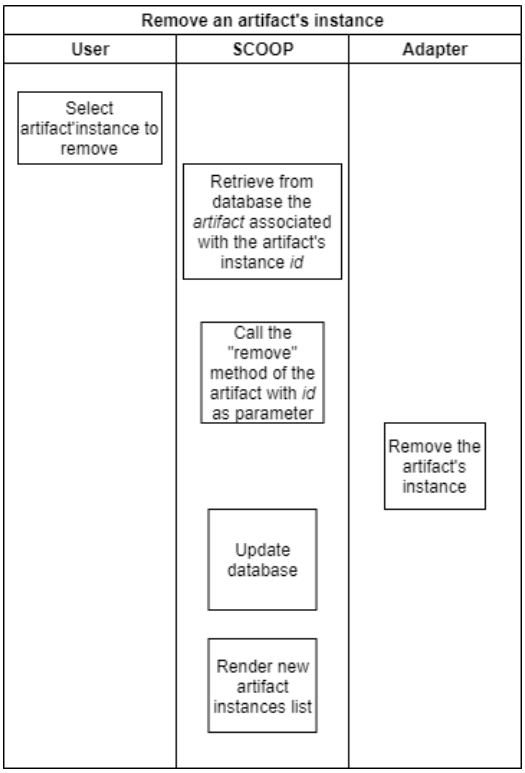

Figure 4.17: Tasks that the user, SCOOP and the Adapter perform in order to remove a particular document associated with an artifact instance.

Remove a Document The user removes an document by first selecting the latter among the list showed in the Figure 4.6. Once SCOOP identifies the document and artifact instance, it calls the "remove" method of the latter. SCOOP updates the artifact instances database accordingly and renders the new list, as shown in Figure 4.17.

Build a test script When the parameter's type is ContractEntity, ConstractInstance or ContractMethodCall the schema, and therefore the form, is built in a slightly different way. This is due to the fact that, to these parameter types, is associated a specific behaviour. In particular:

- A ContractEntity parameter's values are the names of the smart contract entities, as to say, the name of the contract's class.
- ContractInstance refers to the instances of smart contracts, thus deployed versions of the smart contracts. This parameter type is generally associated with the addresses of the deployed contracts.

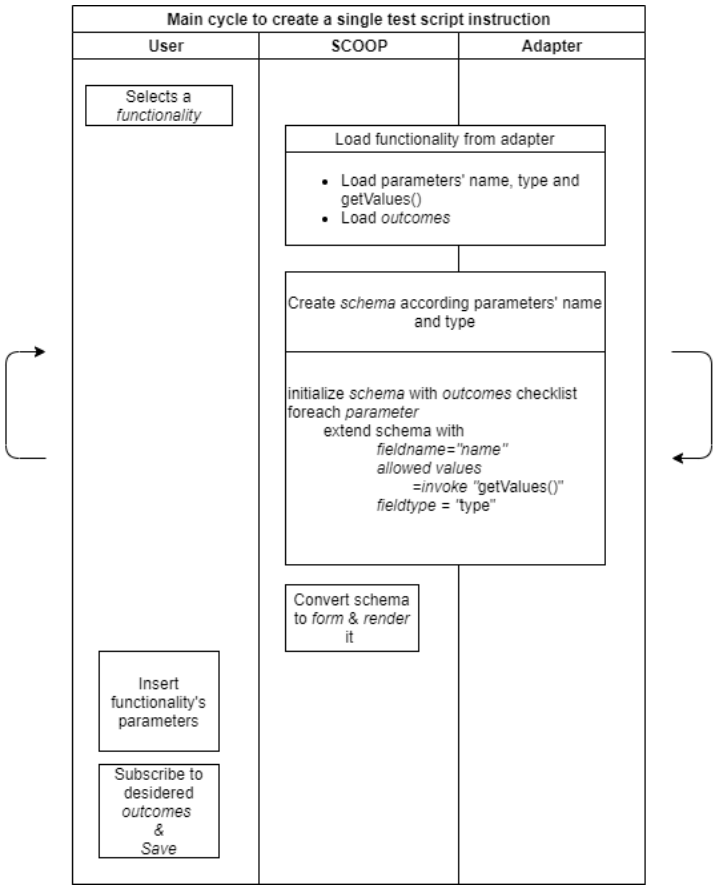

Figure 4.18: Main sequence of events regarding the cycle for building a test script. This diagram does not take into account that the parameter type may be non-primitive, involving instead a smart contract entity, instance or method call.

- Supporting the type ContractMethodCall implies displaying a dynamic form that allows the user to select a contract instance, the list of methods of the contract instance and the list of parameters for each method.

We derive that handling the first two cases is quite straightforward since their allowed values still consists in a simple list. The ContractMethodCall case is more complicated, in particular we find that some fields depend on other fields. For instance, after selecting a contract instance method, the form will display all of the parameters of that method, if the user changes method choosing another, the form will be updated to display the parameters associated with the new choice.

For each of the types listed above, `SCOOP`'s `GUI` is able to render the

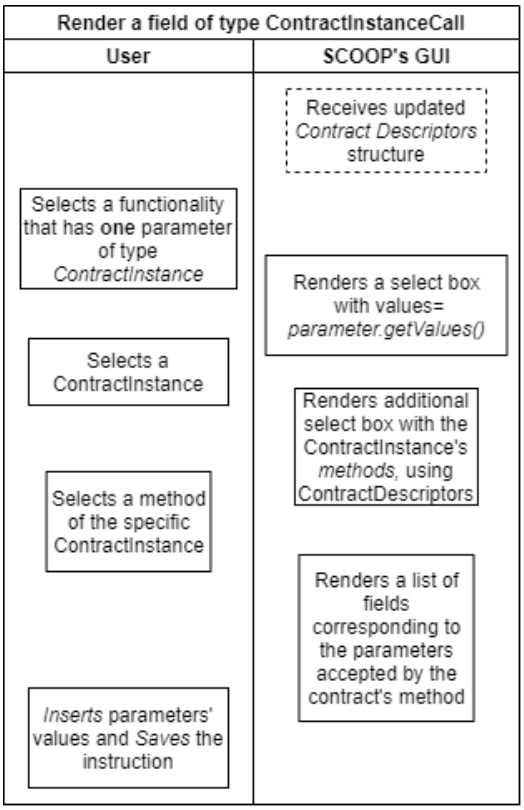

Figure 4.19: This figure describes how a user interacts with SCOOP when *he selects a functionality whose parameter is of type ContractMethodCall. For simplicity, we assumed that the latter is the only parameter, since in any other case fields are appended to the form, as it has already been shown.

corresponding fields using the "Contract Descriptors" structure.

Run a test script Once the user creates a testing script, he can run it by selecting it among the list of saved scripts and subscribing to one or more outcomes the functionality will eventually produce as output. Each instruction of the test script corresponds to a functionality, which is invoked by SCOOP and executed by the adapter. The functionality may, in turn, invoke one or more of the framework's commands.

While the functionality is being executed, it may print additional logging information to the log monitor that SCOOP provides. Finally, the functionality returns the list of all the possible outcomes specified in the functionality object. SCOOP will therefore filter the produced outcomes by showing the user just the ones he previously selected. This process is summarized in Figure 4.20.

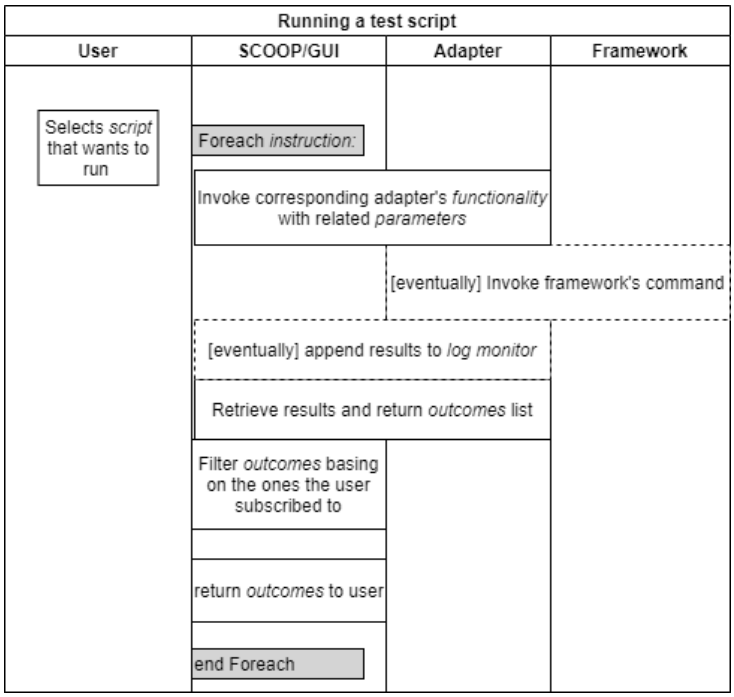

Figure 4.20: To each single instruction of the test script saved by the user corresponds a functionality exposed by the adapter. After invoking the functionality, the adapter may or may not invoke framework's commands. When the functionality returns, logging information may be sent to the log monitor and the outcomes produced by the functionality will be displayed to the user.

4.5 Common Used Terms

The following is a list of definitions for the main terms used in this chapter.

(Smart contract prototyping) Framework: any framework that implements one or more stages of smart contracts' `DevOps` cycle explained in Section 2.2.4.

(SCOOP's) API : `SCOOP`'s Application Programming Interface. Its role is to provide, to the tools underneath the `SCOOP` level, a protocol to generate the ad-hoc loads compatible `SCOOP`, analogously to what happens with BCTMark in Section 3.1.4.

(Sub-Module's) Adapter : the Adapter is the result of the developer implementing the `SCOOP's API`. The Adapter therefore determines ad-hoc loads exposing them to `SCOOP` via its `API`.

Sub-module : the sub-module is an abstract concept, it is the unification of the adapter and the framework(s) the adapter communicates with.

Document : a document is any kind of information the sub-module needs users to provide, in the format and syntax the sub-module expects it to be.

Artifact class / Artifact abstration : is the generalization of the artifact instance. It answers the question: which properties and assumptions can `SCOOP` make about **every** document **any** sub-module might need the user to provide?

Artifact instance : an artifact instance describes properties relative to a **particular set of documents** or entities a sub-module might need. For instance, if every network used by the sub-module must have a name, the network artifact instance is an object that specifies this requirement.

Schema : it can be thought of as a form. Describes the syntax of documents and information with a key-allowed values data structure. Each artifact instance owns its own schema, each schema maps out with exatcly one form and each form, intuitively may produce several results.

Parser : transforms the output of a form into a document or information that can be directly manipulated by the sub-module.

Reviver : the inverse function of the parser, it transforms a document or information used by the sub-module into a key-value data set used to initialize a form so that the user can directly manipulate the data-set.

Test Script : it is an ordered sequence of actions that the user requires the sub-module to perform, using `SCOOP` as an interpreter.

Test script interpreter : `SCOOP` component that handles the interpretation of the high-level test script written by the user into the low level sub-module's actions

Smart Contract Abstraction / Contract Descriptor : an always up-to-date data structure that encapsulates the properties of smart contract entities living in the sub-module's environment. Properties such as the methods of the smart contract and their parameters.

VMAdapter : the `SCOOP`'s `API` component that, once implemented by the adapter, generates an ad-hoc load that `SCOOP`'s back-end uses to keep the ContractDescriptors up-to-date.

Chapter 5

Implementation Skeleton

In this chapter we cover and expand the critical points encountered creating the building blocks of each level of SCOOP's architecture. In Section 5.1, we provide a description of the runtime environment, as well as the frameworks and libraries used. This will give us the necessary notions to discuss about SCOOP's API and all of its relevant parts in Section 5.2. We will then move up to the back-end in Section 5.3, discussing the main tasks performed by this component. Finally, in Section 5.4 we discuss the user interaction design.

5.1 Languages and Platforms

Languages and platforms used to build an application significantly shape what tasks the application can perform. If the technology stack underneath the application is well chosen, there is also a saving in terms of time and effort that the developer puts in building the application.

Considered this, we present in Section 5.1.1 our runtime of choice, in order to introduce the frameworks and libraries supported by the latter in Section 5.1.2. Given the complexity of each library and framework, every element of this section will be introduced delivering its key features on a need to know basis.

5.1.1 Runtime Environment

Our runtime of choice is Node.js for three main reasons:

1. It allows coding both front-end and back-end using the same language, as to say, JavaScript.

2. It provides natively a package management system called NPM (Node Package Management).

3. It allows building fast and scalable network applications thanks to non-blocking I/O calls.

(1): JavaScript

Using the same language for both front-end and back-end turned to be a convenient choice since it helped softening the learning curve. In addition, JavaScript is a very popular language with a particularly vast support community, this results in a large amount of dedicated libraries and frameworks. It is worth pointing out that it is an interpreted and weakly typed language, features that generally cause several run-time errors. We address this issue in Section 5.1.2.

(2): Node Package Management

The `NPM` technology allows developers to create packages that can be stored publicly in repositories. In addition, it adds the version and inter-package dependency control features. This is important for our purposes since `SCOOP` is a modular application, and therefore a technology like `Node.js` that natively supports packages intuitively seems to be appropriate. In fact, `Node.js` associates to every Javascript file within the environment the special object "module.exports", which is used to expose one Javascript file's properties to all the others. In particular, a Javascript file that "requires" another one, has access to the latter's "module.exports" object.

(3): Non-blocking I/O

The `Node.js` environment supports non-blocking calls, and since we already anticipated that `SCOOP` needs to interact constantly with external tools, a mechanism that avoids blocking the entire `SCOOP`'s environment waiting for external tasks to terminate certainly is a good feature.

5.1.2 Frameworks & Libraries

In this section we are going to introduce the three key tools that helped building both `SCOOP`'s front and back-end. In particular, we address the portability issue introducing Electron and begin to approach `GUI` design using React. The last point of this section aims to solve JavaScript's weakly typing issue, introducing a new syntax: Typescript.

Electron

Electron allows to partition the application in two different processes, the main process and the renderer process. These two type of processes run in separated `Node.js` environments and communicate using two different libraries that implement an `IPC` (Inter-Process Communication) paradigm, using pipelines[24].

Portability among different operating systems is an important feature for `SCOOP` and Electron is, to the best of our knowledge, the best alternative when it comes to build cross-platform desktop apps, since it allows to easily wrap the entire application in several different types of executable files.

In particular, using Javascript, `HTML` and `CSS` the developer is able to build an application just like *he would do in a web environment. This can be of great help when the idea is to include a `GUI` within an application, as it happens with `SCOOP`.

Having two different processes guarantees a better separation of concern between grahical and non-graphical tasks. In particular, the renderer process is responsible for running the `GUI`, which is a full-fledged web page. The main process manages the main windows of the application, sending events to the latter and controlling the life cycle. In is worth noticing that the `IPC` system requires serializable data, this is of particular relevance for Section 5.2.5.

React: Components & Hierarchy

React is a JavaScript library that facilitates designing user interfaces[25]. It does so by enhancing the classic tagging system of meta-languanges like `HTML`. The central concept of React development and thinking is the Component.

In general, a React Component implements an interface that contains the "render()" method, the render method returns in turn a component that needs to be rendered. A Component object is the conceptual equivalent of any `HTML` or `XML` tag, in fact it accepts properties, namely *props*, and may display visual information that is a byproduct of those same props. When props change, the component is automatically re-rendered.

The language in which Components are expressed is `JSX` (JavaScript Serialization to `XML`)[26], which is functionally equivalent to JavaScript, although its syntax is a mix of JavaScript and markup languages. The "render()" method will therefore return a `JSX` object.

The interesting aspect about Components is that they can be both stateful and stateless, respectively when the Component type is a Class Component and a Functional Component. The possibility of adding a state to a Component adds an incredible amount of expressive power to the traditional

markup system.

A Component may return other components, its children, within its structure, so that when the component is re-rendered all of its children will be too and not viceversa.

Another important paradigm concerns the visibility of a component's state. In fact, a component can only visualize its state and pass props to its children. It often happens that a child needs to update its parent's state. Since the parent is only allowed to pass props to its children, the parent creates an ad-hoc method that updates its own state and then passes this method to its children using the props. This way children will update the parent's state safely, calling a method provided by the latter. This paradigm is shortly referred to as "lifting the state up".

Those mechanisms are the heart of React programming and allow having a hierarchical, modular structure within the application. Of course, the possibility of creating custom modules adds value to this framework too, since developers are now able to build upon this mechanism in order to automize all sorts of tasks, as the next section shows. A special case is the one in which a component needs to render a list of components. In this case, React allows to render an array of components within the render method.

Uniforms

Uniforms is a framework based on React that allows to easily build and style forms[27]. React allows to create and style forms exactly as the developer would do with traditional `HTML` and `CSS`, although Uniforms gets nearer to the `SCOOP`'s purposes by allowing to create forms using schemas. This is a much more convenient system, since any developer (or even an unexperienced user) can understand the structure of a schema without studying the `HTML` syntax from scratch. In addition, Uniforms allows the developer to:

- Create custom fields.
- Easily set-up required fields.
- Easily interpret forms' results using `JSON` syntax.

TypeScript

JavaScript code can crash at runtime since there is no type-inference mechanism. Code editors usually take advantage of this mechanism to help the developer lower the number of mistakes caused by distraction when writing code. In addition, a strongly typed language helps in the developing process,

since it supports classes and object, tools that have been proven to enhance computational thinking processes.

TypeScript offers critical features such as conditional types, interfaces, maps, run-time type checkers and so on. The most important fact is that TypeScript is a new syntax for Javascript, not a language that requires a different interpreter[28]. TypeScript syntax is directly transformed to the Javascript's one by using the TypeScript Compiler. We derive that TypeScript code runs on any platform Javascript's one does.

5.2 SCOOP's API

Since `Node.js`'s packages can export properties using the "module.exports" object, this first version of `SCOOP`'s `API` expects a module to expose the `API`'s implementation properties using this mechanism. We therefore assume that all the objects in the following sections have been properly exported.

5.2.1 Artifact Class

This section discusses two essential facts about artifacts. First, we remember that, in `SCOOP`'s view, an artifact is a class that models useful properties about any kind of structured document the user needs to interact with during the prototyping process. The interaction is delivered using forms, expressible as schemas.

One of our aims is to use schemas to build forms automatically. Thus, in the following section, we introduce a library for React that converts schema into forms.

Another useful property of an artifact is whether it is required or not and whether it is iterable or not. We remind that the "iterable" property refers to number of possible instances of an artifact class. In particular, it tells `SCOOP` whether that number is limited or not.

In general, the structure of an artifact instance differs from the form's. In the second part of this section, we address this issue.

Uniforms Schema

In Listing 5.1 we find an example of schema and that can be correctly transformed into a form. Uniforms introduces the "type" field in order to distinguish between fields that must be rendered as simple text boxes and fields that accept numbers.

```
schema :
```

```
{
        title: 'Network',
            type: 'object',
            properties: {
                network_name : {type: 'string'},
                host : {type: 'string'},
                port : {
                    type : 'integer',
                    minimum: 55000,
                    maximum: 65000,
                },
                network_id : {
                    type : 'string' ,
                    description: 'Use * to match any id'
                }
            },
            required: ['network_name']
}
```

Listing 5.1: An example of schema compatible with the features provided by Uniforms.

The form corresponding to Listing 5.1 is displayed in Figure 5.1, we notice that the field "network_name" becomes red when the user submits the form without inserting it first. This is due to the fact that we marked the network name field as "required". In addition, the field with type "integer" also displays arrows to increment and decrement the value, which also has a minimum and maximum bound.

When the form in Figure 5.1 is submitted, the generated result is a model containing the information inserted in the form. The model corresponding to the form above with is shown in Listing 5.2.

```
const model = {
  "network_id": "*",
  "port": 55000,
  "host": "127.0.0.1"
};
```

Listing 5.2: The model object. It is generated when the form is submitted.

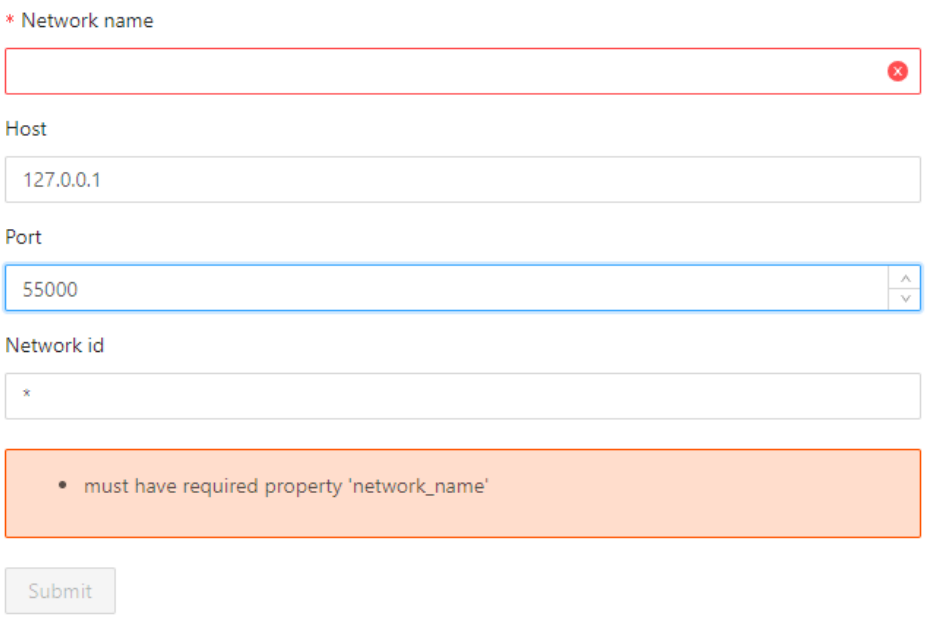

Figure 5.1: Form corresponding to the schema in Listing 5.1

Parser & Reviver

When the user wants to add an instance of artifact, SCOOP retrieves the schema and artifact name from those exposed by the adapter. After the user selects which artifact *he wants to instantiate and SCOOP renders the related schema, the form is compiled and submitted, this results in a model as reported in the previous section.

Parser's task is to take as input the model an generate the artifact's instance, assigning an id to the artifact instance and returning it.

In the GUI, the artifact's instances' identifiers are displayed with the pattern **artifact class name*_*id**, so that the user can distinguish between different instances of different artifacts.

When a user wants to edit an artifact instance, *he will select the desired artifact choosing among the list, whose elements respect the pattern of artifact class name and id. SCOOP will then call the artifact's class reviver method with the id in the artifact's instance name as parameter. The output of the reviver is a model identical to the one originally submitted by the user. That model is used to initialize the form correctly.

5.2.2 Param & Functionality class

The Functionality objects exposed by the adapter respect the guidelines expressed in Section 4.3.3. In Listing 5.3 we find the specification of a Functionality; it's worth noticing that outcomes are thought of as strings and that the parameters accepted by the Functionality are expressed as an array of ParamDescription. The specification also includes the "environment" object the "functionality" uses to look up environmental variables during the execution.

```
type Functionality = {
    functionName: string,
    functionality : Function ,
    paramValues : ParamDescription[],
    outcomes : string[] | undefined,
    environment: Object
}
```

Listing 5.3: The functionality's type.

Parameters' names, types and allowed values are modelled through the ParamDescription type, specified in Listing 5.4.

The "getValues()" method is either an anonymous function that returns the list of allowed values or a null object, for those parameters that accept any value.

```
type ParamDescription = {
  name : string,
  type: string,
  getValues : ( () => string[] )| null
}
```

Listing 5.4: The functionality's parameter type.

A Functionality exposed by a certain adapter must therefore respect the specification above in order to be considered valid by `SCOOP`.

5.2.3 Environment

The concept of environment in the implementation is more vast than anticipated in Section 4.3.3. In fact, not only it is necessary to expose a global environment object within which executing all the functionalities that do not specify a local one, but we also need an environment for the scripts executed during the workflow. For this reason, we extend the `API` with an object that

indicates the working directory (`WD`) of the sub-module, thus the one in which commands will be executed. The "module.exports" object will therefore own an additional property "WD".

5.2.4 Scripts & Workflow

Associating scripts to different stages of the lifecycle is necessary in order to automatize several actions that would otherwise need to be repeated by the user or implemented by the developer. In particular, scripts are associated to:

- Start and the end of a sub-module's life cycle / start script & end script.
- The moment prior to the execution of a test script / pre-test script.
- The moment subsequent to the execution of a test script / post-test script.
- The removal, addition and update of an artifact instance / update script.

The adapter exposes scripts by adding to the "module.exports" object properties whose names correspond to the scripts listed above.

5.2.5 VMAdapter & Smart Contract Abstraction

In Section 4.3.3 we discussed about the `SCOOP`'s smart contract abstraction, identifying it with a data structure named ContractDescriptors. We derived that our `API` needs another key component in order to build that data structure, the `VMAdapter`. This component's behaviour is defined by the adapter's developer and allows `SCOOP` retrieving and interpreting the `ABI`s of smart contracts in order to obtain the ContractDescriptors structure.

In this implementation, we simplified the `VMAdapter` merging the methods getABIs() and parseABIs() into a method called "getContractDescriptors". We provide an example of implementation of this method for the combination of both the Truffle environment and the Ethereum Virtual Machine `ABI` syntax.

First of all, let us show in Listing 5.5 the specification of a single ContractDescriptor:

```
type ContractDescriptor = {
    contractName : string,
    methods : Map<string, Map<string, string>>;
```

```
}
```

Listing 5.5: The structure ContractDescriptors is considered an array of ContractDescriptor objects.

We therefore conceive the global ContractDescriptors structure as an array of ContractDescriptor objects. It is worth noticing that this `VMAdapter` specification does not model a deployed contract as reported in Section 4.3.3, instead it refers to the contract entity, so that the information concerning the contract class will not be replicated in all of its deployed versions.

This implementation identifies smart contract instances by assuming that the `SUT` will probably need to keep track of the latter, for example by storing the addresses of these deployed contracts. We therefore extend the `VMAdapter` by adding a method that returns a structure that maps contract classes to its deployed instances in a given instant.

5.3 Back-end

In this section we analyze key tasks that the back-end performs. In Section 6.2 we explain how the two processes of Electron communicate and in Section 5.3.2 we briefly analyze what this communication paradigm implies. Finally, in Section 5.3.3, we discuss the core of the back-end, namely the test script interpreter, which is the component that actually implements the interaction with the sub-module.

5.3.1 Interaction with the Front-end

As reported in Section 5.1.2, the renderer and main processes communicate using an inter-process communication system based on pipelines. It is possible to send messages using labeled channels, namely events, the sender must publish and the subscriber needs to register to. A usage example of this mechanism is shown in Listing 5.6:

```
ipcMain.on('form_request', (event, request) => {
    var form = getForm(request)

    event.reply(form)
})
```

Listing 5.6: A simplification of how ipcMain publishes an event and responds to it.

We can now formally state that the back-end (risp. front-end) corresponds to the main process (risp. renderer process).

It is important noticing that the mechanism that lies behind pipelines requires data serialization, in fact it is impossible to send raw non-primitive structures such as Maps or Functions. The next section addresses this issue for what concerns the Functionality objects.

5.3.2 Data Serialization

In Section 5.2.2, both Functionality and Param specifications include non-primitive types. Since the renderer process needs to display functionalities to the user, these data need to be sent to the renderer process. We therefore need to serialize the data, this implies resolving the non-primitive types in specifications.

In this case, the only types that must be resolved are the Function objects "functionality" from FunctionalityDescription and "getValues" from ParamDescription.

As shown in Listing 5.8, the fields "functionality" and "environment" are directly removed, since the renderer process does not need to execute the functionality. The "getValues" property from ParamDescription can be serialized by simply invoking the anonymous function that is supposed to retrieve the allowed values of the parameter, if present. Values returned by the function will be pushed within an array, as shown in Listing 5.7.

This way, the whole SerializableParamFunctionality object, containing the SerializableParam, can be serialized.

```
type SerializableParam = {
    name : string,
    type : string,
    values : string[] | null
}
```

Listing 5.7: The serializable version of ParamDescription type.

```
type SerializableFunctionality = {
    functionName: string,
    paramValues: Param[],
    outcomes: string[] | undefined
}
```

Listing 5.8: The serializable version of FunctionalityDescription type.

5.3.3 Test Script Interpreter

The test script interpreter receives as input an ordered array of instructions from the front-end. To each instruction corresponds, as output of the interpreter, an array of outcomes which are the, in turn, the output of each functionality, filtered basing on the outcomes the user is interested in receiving. A single instruction must be serializable and contain information about:

- The identifier of the Functionality selected by the user.
- An array of the Functionality's parameters.
- An array of objects that describe which outcomes the user wants to receive.

As pointed out in Section 4.3.3, a parameter may be of type ContractInstanceCall and therefore must contain the following information:

- Deployed contract identifier.
- Contract method name.
- An ordered array of method's parameters.

In Listing 5.9 we show how a single instruction, containing a parameter of ContractInstanceCall type, appears. In this example, the functionality that has been selected allows the user to invoke a contract method a given number of times.

```
{
  functionality: multipleContractMethodCall,
  params: [
            {
              contractID: SimpleStorage_1,
              methodName: setValue,
              methodParams: [
                              { value:1 }
                            ]
            },

            {
                numberOfInvocations: 1000
            }
```

```
            ],

  outcomes: [
              {
                  totalLatency: false
              },
              {
                  throughput: true
              }
             ]
}
```

Listing 5.9: JSON representation of a single instruction within a test script.

The type Function allows to apply a given function to a list of parameters and to associate the execution of the function with an environment object. This is exactly the start point of the test script interpreter.

In Listing 5.10, we provide a simplified version of the testscript interpreter.

```
var functionalitiesOutcomes : Array[]<Object> = new Array()

/* Instructions come from the renderer process and are
   determined by the user                                    */
for (var instruction in instructions) {
  /* Retrieve from the adapter the Functionality
     named within the instruction object                     */
  var functionality : Functionality
    = getFunctionalityFromName(instruction.functionality);

  // Extract the body and environment of the functionality
  var body = functionality.body;
  var environment = functionality.environment;

  /* Apply the body of the function to the parameters
     specified in the instruction object
     executing the functionality in its environment */
  var outcomes : Array[]<String>
    = body.apply(environment, instruction.params);

  /* Filter the outcomes obtained by the execution
     basing on the user's preferences stored
```

```
      within the instruction object                    */
  var filteredOutcomes =
      outcomes.filter ( (outcome) =>
                              {
                                var selectedByUser : Boolean =
                                 instruction.outcomes.contains(outcome);
                                return selectedByUser;
                              }
                          );
  // Store the outcomes in the array
  functionalitiesOutcomes
         .push(
                {
                  functionality: instruction.functionality,
                  outcomes: [...filteredOutcomes]
                }
               )
}
return functionalitiesOutcomes;
```

Listing 5.10: How the test script interpreter executes functionalities. The "instructions" array contains several instructions formatted as in Listing 5.9.

The output of the test script interpreter is a serializable, ordered array of objects, where each object contains the name of the functionality and the outcomes requested by the user.

5.4 Front-end

We focus on adding artifacts and executing one test script at a time using two different components in the hierarchy: the FormDisplay and Command Array components.

When the user clicks on a tab, the ipcRenderer process communicates the decision to the renderer process, which in turn sends the information that needs to be rendered accordingly. As Figure 5.2 shows, the front-end is composed by three main parts, as to say, the Renderer, FormDisplay and CommandArray components. In Sections 5.4.1 and 5.4.2, we observe that the Renderer and FormDisplay components have a simple structure, whereas the CommandArray component described in Section 5.4.3 has a nested, matryoshka-like structure.

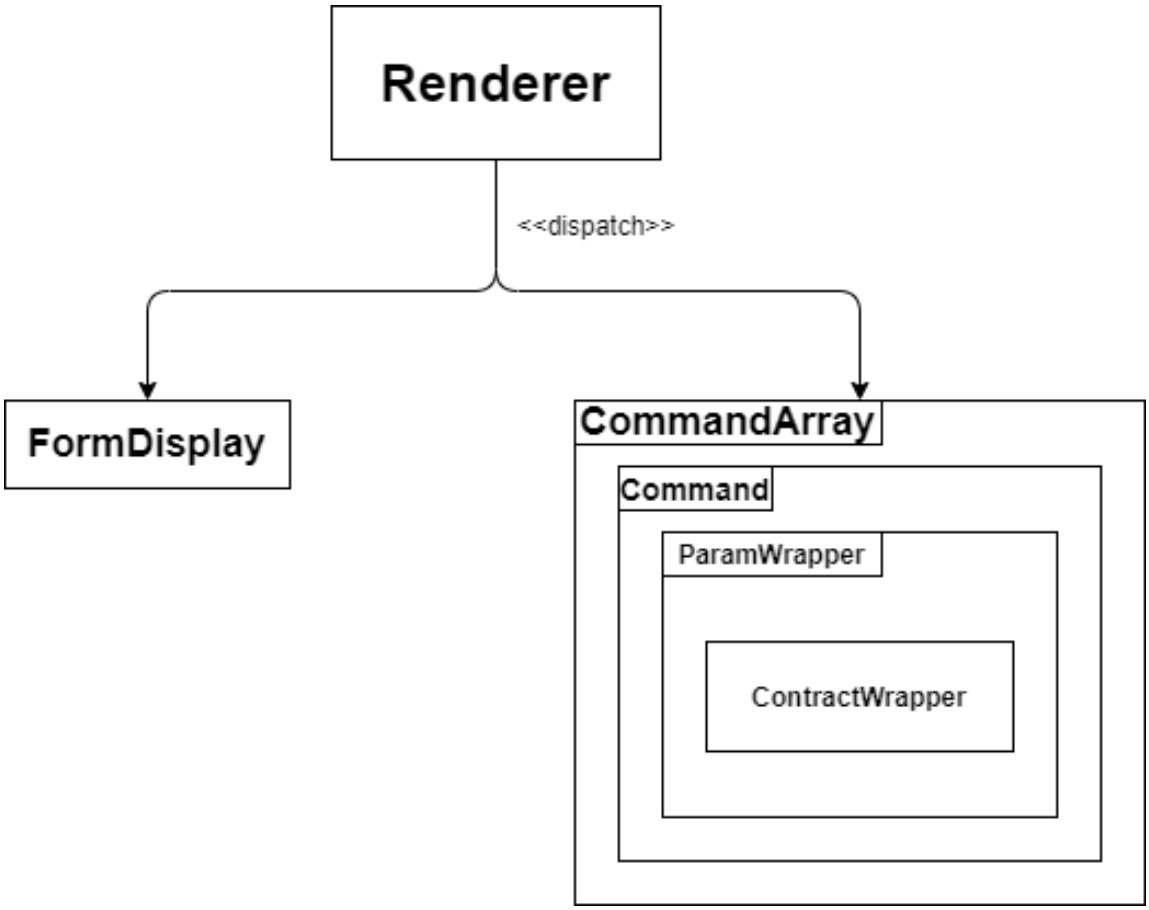

Figure 5.2: A global view of the components' hierarchy.

5.4.1 Renderer: a Component Dispatcher

The renderer component occupies the highest position in the hierarchy. It has the job of dispatching the messages coming from the main process using the ipcRenderer. In particular, the messages coming from the mainProcess, in response to the user selection, determine if the Renderer will display a FormDisplay or CommandArray component. Afterwards, the renderer process sends the data provided by the user to the main process, which will treat the data according to the type of request. The two components dialogue with the main process differently, as we show in Sections 5.4.2 and 5.4.3.

5.4.2 FormDisplay

This component is the simpler one. In fact, when the user requests a form, the renderer waits for its schema to be sent and subsequently renders the

FormDisplay. In turn, FormDisplay uses the Uniforms library to convert the schema into form automatically.

When users submit forms, the result is a "model" object containing the submitted information. Model objects are sent through the `IPC` system, afterwards the main process infers which artifact the models refer to and interprets them accordingly via the artifact's parser method.

5.4.3 CommandArray Hierarchy

When the user's choice is to build a test script, *he must be able to add, edit and remove instructions from the latter, with respect to the functionalities exported by the adapter. This is our main focus. Furthermore, a user must be able to open the scripts he previously created, in order to edit or execute them.

Figure 5.3 shows an example of how different components operate within the `GUI` users interact with in order to create a test script. In this case, the user wants to create a script with two instructions. The first deploys the project onto the Ganache network, the second ("methodCall") lets the user choose a deployed version of a smart contract and call one of its methods ("setValue") a certain number of times.

In general, the hierarchy, in a top-down order, consists in the following components:

1. **Renderer:** decides whether a CommandArray or FormDisplay component will be displayed or not.

2. **CommandArray:** stores a list of Commands and lets the user add, remove instructions from the scripts, as well as executing it.

3. **Command:** encapsulates the adapter's functionality selected by the user and the ParamWrapper.

4. **ParamWrapper:** displays a list of textboxes corresponding to the functionalities parameters and their accepted values. If one or more of the parameters' type is ContractCallInstance, it renders the ContractWrapper component.

5. **ContractWrapper:** uses the ContractDescriptors structure to let the user choose a deployed contract, as well as its methods and parameters.

Each component is child of the upper one. Let us discuss each component separately.

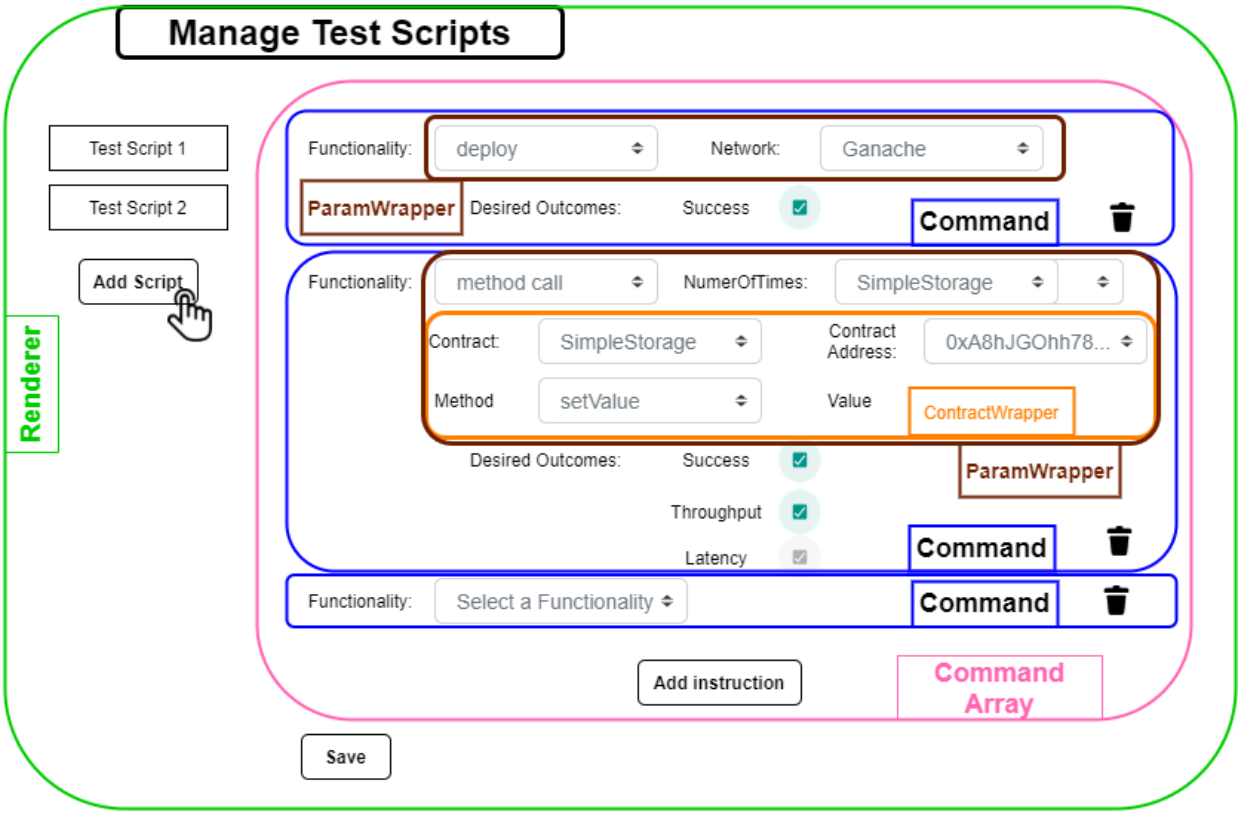

Figure 5.3: A view that merges how the build testing script stage is displayed to the user and which components the same user interacts with. The user can also remove instructions from the script.

(1): Renderer When the user's request is to build a test script, the Renderer gives the control to the CommandArray component, which is one of its children. The props that Renderer gives to CommandArray in input are the Functionality list and the ContractDescriptors.

(2): CommandArray CommandArray has the task of:

- Storing within its state the list of instructions added up until that moment by the user.
- Displaying the instructions list.
- Allowing the user to add an instruction or executing the script using buttons.

The global information on the test script is stored in this component, and it is updated whenever a user causes changes in the test script. Each instruction is modelled by a different Command component, which is a CommandArray's children. In particular, the test script data structure is updated when an instruction is added or updated by the user. The global test script must therefore change accordingly. The first case (add an instruction) is easy to handle, since CommandArray only needs to add an entry to its test script data structure. The other cases are more difficult to handle, since children should be able to produce changes in the CommandArray global structure but they cannot access directly its state. As explained in Section 5.1.2, we get around this problem by "lifting the state up", therefore passing the children the methods necessary to update the parent's state. When a change is produced within a CommandArray heir, there will be a cascade of calls that propagates up to the CommandArray component.

(3): Command A Command component encapsulates a select-box that allows the user to select a functionality. When a functionality is selected, this component will automatically display the functionality's parameters as text boxes within the ParamWrapper component. Command updates CommandArray's state whenever a user selects a functionality for the first time or changes his/her selection. In particular, if a user modifies the content of the "functionality" text box by selecting another, the Command component calls the method "updateCommand" in its props. The "updateCommand" method is executed in the context of the parent CommandArray and has the effect of updating the entry of the global test script structure in such a way that it will contain the newly updated functionality.

ParamWrapper may need to update, within the CommandArray's test script data structure, the parameters associated with the selected functionality. Nevertheless, ParamWrapper has the task of displaying parameters and letting the user update them, it does not need to know the functionality those parameters refer to. Command will therefore pass to ParamWrapper only the information needed to display the parameters, in order to introduce a better separation of concern between components.

In the same way, if ParamWrapper component is updated by the user, it does not need to communicate directly with CommandArray. Instead, it communicates to Command the newly updated parameters via the method "updateParams". At this point, Command propagates the request to its parent CommandArray, which now has the necessary amount of information to identify the instruction in question and update its selected method and parameters.

In the next paragraphs, we will refer to the mechanism above as "partial evaluation". In general, the parent's information does not concern the child, but is does concern the global test script structure. So, when a change is produced in a component, it will call a method that has been partially evaluated by his parent to include the same parent's information. This mechanism repeats itself over the hierarchy up to the CommandArray component.

(4): ParamWrapper ParamWrapper receives via its props the partially evaluated method "updateParams" and the list of the parameters of the Command component's selected functionality, together with the allowed values and type of those parameters, if they exist.

If the parameter's type is primitive, ParamWrapper directly displays a labeled text box with the allowed values of the parameter as possible values. Else, if the type is "ContractCallInstance", ParamWrapper will render a ContractWrapper component. ContractWrapper receives as props the ContractDescriptors and a partially evaluated method that will allow the ContractWrapper to lift the state up to the CommandArray component. This will happen whenever a contract method is selected or that precise method's parameters are inserted.

(5): ContractWrapper This component receives the ContractDescriptors structure and interprets it in order to let the user:

- Select a deployed contract identifier.
- Select a method of that specific deployed contract.
- Insert the contract method's parameters using checkboxes.

Whenever a select box or text area's content changes, the ContractWrapper component will lift the state up to the CommandArray component in order to update the global test script structure, that describes all of the instruction that compose the script.

Chapter 6

Use Case: Creating a Sub-module

In this chapter we show a possible use case of the implementation described in Chapter 5. In particular, we use `SCOOP` to allow inexperienced users to use the Truffle framework of the Truffle Suite presented in Section 3.1.1, and studied in more detail in Section 6.1.

To do so we implement a sub-module that allows the user to add, remove and edit smart contracts from/to a Truffle project as well as to compile and deploy them onto a network of choice using a custom compiler version.

In Section 6.2 we give an intuitive, global view of how the user interacts with the system and viceversa.

6.1 Deepening Truffle

Two commands of Truffle are made available to the user:

- **compile**: compiles all of the smart contracts' source codes stored within the default "Contracts" directory, using the compiler version specified in Truffle's configuration file. The output is a "build file" that is placed in the default "Build" directory.
- **deploy** [**network**]: deploys contracts onto a network of choice following the migration scripts in the "Migrations" directory. If not specified, Truffle will try deploying on a default network. The main output of this process is a deployed address. In addition, this commands compiles the smart contracts that still have not been compiled.

To be more thorough, let us analyze in detail some of these key Truffle components:

- **Truffle's configuration file** stores metadata such as the custom networks the user adds and the compiler of choice. It is due noting that the user can only specify one preference of compiler.
- A smart contract's **build file** is created when the latter is compiled and contains the smart contract's `ABI` along with information useful for Truffle, such as the smart contract's source code and bytecode. Build files also contain metadata about the deploying stage, such as the address of deployed contracts.
- **Migration scripts** specify which smart contracts the user needs to deploy and in which order. This sub-module assumes that every smart contract in the project needs to be deployed. An example of migration file is reported in Listing 6.1.

```
// Import contract data from source files
var SimpleStorage = artifacts.require("./SimpleStorage.sol");
var OtherContract = artifacts.require("./OtherContract.sol");

module.exports = function(deployer) {
    /* Deployer is the Truffle wrapper for deploying
       contracts to the network                      */

    // Deploy the contract to the network
    deployer.deploy(SimpleStorage);
    deployer.deploy(OtherContract)
}
```

Listing 6.1: A typical migration file. To deploy a contract, Truffle provides the 'deployer' wrapper.

The following two additional Truffle's commands will be used by the adapter and not directly by the user:

- **init** initializes Truffle's directory and environment with the "Contracts" and "Migrations".
- **clear** clears the Truffle environment deleting the contents of "Contracts" directory and the entire "Build" directory, restoring the migration script.

The "init" (resp. "clear") command is executed after (resp. before) the sub-module is loaded (resp. closed).

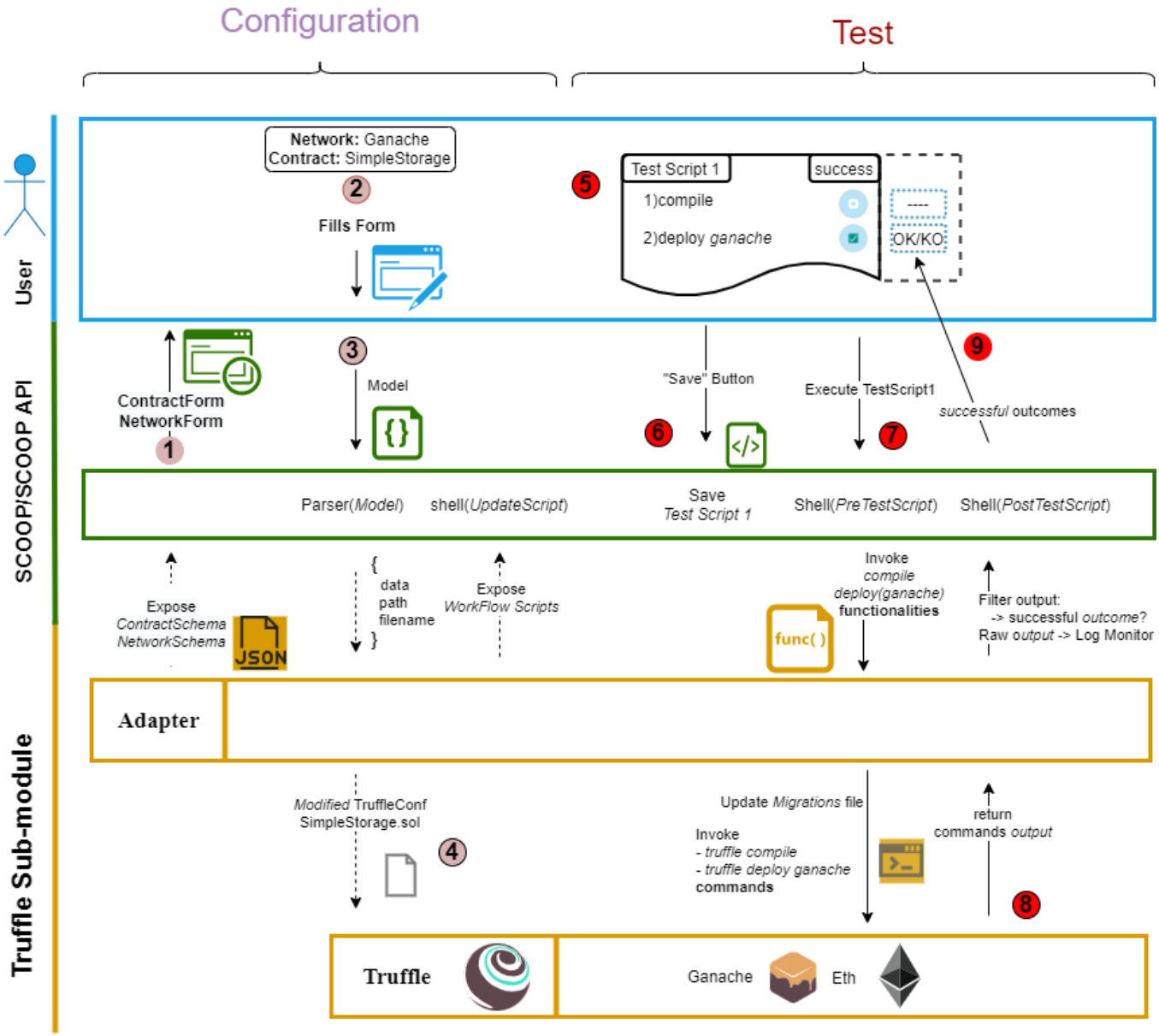

Figure 6.1: The two macro-steps are Configuration and Test. In the configuration, the user requires the form associated with the Network and Contract artifact instances and insert data that will be used in the test stage. Afterwards, the user designs a test by creating a test script and choosing which outcomes *he is interested in receiving. In this case, the only outcome each functionality provides is the success flag.

6.2 Interaction Example

The purpose of this section is to show how the user, SCOOP and our sub-module interact, considering a particular sequence of actions performed by the user.

In particular, the user wants to:

- add a network and a smart contract to the workspace, respectively the network metadata associated to Ganache framework presented in Section 3.1.1 and the SimpleStorage contract reported in Section 2.2.2.
- create a test script containing two instructions, one to compile the entire project and the other to deploy all the contracts.
- execute the script and visualize the outcomes.

Let us comment each step of Figure 6.1.

1. Once the adapter exposes to the SCOOP API the schemas related to the network and contract artifact instances specified in Section 6.3.5, SCOOP can render a form according to the users choice.
2. The user fills the form with the proper information and submits it.
3. Once the user submits the form, a model object corresponding to the form's result is generated. SCOOP uses the artifact instance's parser defined in Section 6.3.5 to process the model.
4. The processed model is used by the parser to update the TruffleConfig file, thus adding a network to it. Since an artifact has been modified, SCOOP executes the UpdateScript exported by the adapter. The behaviour of the workflow scripts will be properly described in Section 6.3.3.
5. The user now decides to create a test script. The first line of the script refers to the compile functionality, the user decides to not receive the outcome of this instruction. The second line aims to deploy the SimpleStorage contract on the local blockchain Ganache. Both compile and deploy functionalities' behaviours will be thoroughly expanded in Section 6.3.6.
6. The user saves the script and SCOOP stores it in its local environment.
7. The user chooses to execute the test script. In response, SCOOP runs the PreTest Script and subsequently uses test script interpreter to evaluate the script itself.

8. The functionalities' behaviour in this case is to send the commands' raw output to the log monitor, that we concretely define in Section 6.3.2. Once the commands and the functionalities return the outcomes, SCOOP retrieves their outputs and filters them, basing on the user's preferences in the checkboxes.

9. The user visualizes each instruction's outcome next to the *success* checkbox, if and only if the latter is checked.

6.3 Creating Sub-module's Adapter

We create an adapter by implementing SCOOP's API, whose diagram is reported in Figure 4.10. A particular case is the Log Monitor, where the adapter's developer only needs to subscribe to the monitor itself, in order to send updates to it.

Let us address each part of SCOOP's API in the following sections.

6.3.1 Global Environment

In this sub-module, the global environment object is the one within which all of the functions are executed. It contains the state of the sub-module, with the state mainly being a set of associations between artifacts instances and their identifiers. The way in which the global environment object is exposed is described in Listing 6.2.

```
module.exports.globalThis = this
```

Listing 6.2: The globalThis (global environment object) of SCOOP's API coincides with the sub-module's environment.

6.3.2 Log Monitor

The Log monitor, as shown in Figure 4.10, is an object exported by SCOOP's API. In particular, the sub-module subscribes to the published monitor object and sends updates to it.

In TypeScript, whenever a module needs resources from another module that exposes them (publishing), it can import the resources in question (subscription). After importing the Log Monitor object, the user can print updates onto it, as Listing 6.3 shows.

```
import 'SCOOPAPI/logmonitor'

logmonitor.send("Some update")
```

Listing 6.3: An example of how sub-modules subscribe to the Log Monitor (import) and send notifies to it.

6.3.3 Workflow Scripts

All scripts are strings that `SCOOP` later interprets as commands for the shell. Let us describe briefly what this sub-module's scripts do without lingering on syntax's details.

- **Start Script**: initializes a new Truffle project.
- **Update Script**: every time an artifact instance is added or edited, compiles the entire project.
- **Pre-Test Script**: records the timestamp of the instant of its execution. Starts the Ganache local blockchain node, **iff** no other network is specified in the Truffle configuration file.
- **Post-Test Script**: subtracts the current timestamp to the Pre-Test Script one, in order to infer the total latency time of the test script.
- **End Script**: Cleans the Truffle envirnoment by executing Truffle's "clear" command.

6.3.4 `VMAdapter`

Our sub-module does not directly interact with smart contracts, since its purpose is to allow users compile and deploy contracts, jobs that are delegated to Truffle. The user - at this stage of our sub-module's evolution - cannot therefore invoke a given deployed smart contract's method. Nevertheless, we provide in this section an implementation of the `VMAdapter` interface, since it may result useful in future in order to continue this work.

As presented in Figure 4.9, we may need to implement the method "getABIs" and must implement "parseABIs".

Our contracts' `ABI`s are placed within the "build" directory of Truffle. Each build file stores the `ABI` in `JSON` format under the field named "abi".

We notice that both directory and field names fall under the assumptions stated in Section 4.3.3, question Q5.

Thus, to retrieve the ABIs, we do not need to implement the "getABIs" in order to retrieve the ABI's, SCOOP's default behaviour handles this possibility. In particular, SCOOP will read the JSON files in Truffle's "build" directory and, for each file, will retrieve the ABI within the "abi" field.

The "parseABIs" method instead is necessary, since SCOOP needs it to build the ContractDescriptors and does not make assumptions about the ABIs' syntax. We notice that these particular ABIs' syntax are valid for all Ethereum Virtual Machines, so this method could be re-used in other contexts, with a different blockchain platform (running the same EVM) or framework underneath the adapter.

We discussed the purpose of ABIs in Section 2.2.2, showing in Listing 2.2 what the SimpleStorage contract's ABI looks like. As anticipated in that section, some fields like "name", "inputs", "outputs" and "type" are of particularly crucial for us to create the "parseABIs" method presented in Listing 6.4.

```
var parseABIs : Function =
  (abis : Array<Object>) => //returns a list of ContractDescriptor
     {
        // Instantiate the ContractDescriptors array
        var ContractDescriptors : Array<ContractDescriptor>
                = new Array()
        //For each contract ABI
        abis.forEach(
          (abi : Object) => {

              //Init a new ContractDescriptor
              var desc : ContractDescriptor
                = new ContractDescriptor(abi.contractName);

              //For each property of the smart contract
              for ( let property in abi.values() ){

                /*iff it is a function
                        => record its name
                        => infer params names and types
                        => add to descriptor
*/
                if ( property["type"] === function){
```

```
                var functionName : string = property["name"]
                var params : Array<Object>
                  = new Array();
                for(let param in property["inputs"]){
                    params.push({
                        paramName: param["name"],
                        paramType: inferType(param["type"])
                      })
                }
                desc.addSignature( functionName , params )
              }


            }
            ContractDescriptors.push( desc )
        }
      )


    return ContractDescriptors
  };
```

Listing 6.4: Our "parseABIs" method for `EVM`-compatible `ABI` formats.

6.3.5 Artifacts

The user is interested in giving three kinds of data to the sub-module:

- **smart contracts**' source codes.
- metadata of **networks** compatible with Truffle onto which **he wants to deploy the contracts.
- the **compiler** of choice.

It is worth noting that we separate the network from the compiler despite the fact that they are stored within the same file (the Truffle configuration file). This choices is motivated by the fact that As anticipated, data in support of the prototyping process correspond to artifacts. We therefore associate a different artifact to each of the bullet point list's elements. We therefore instantiate the `API`'s artifact class presented in Figure 4.7 with the smart contract, network and compiler artifact instances in the following paragraphs.

Each paragraph contains six listings representing the code corresponding to the Artifact's class fields: Schema, Parser, Reviver, Remove, Required and Iterable properties.

Each schema will be added to an array that is exposed using the module.export object, as Listing 6.5 shows.

```
var artifact_1 : Artifact =
        new Artifact(
                schema,
                parser,
                reviver,
                remove,
                required,
                iterable);
var artifact_2 : Artifact = [...]

var artifacts : Array<Artifact> =
        new Array(artifact_1, artifact_2)

module.exports.artifacts = artifacts
```

Listing 6.5: How sub-modules expose artifacts once their properties have been defined.

Smart Contract

Schema:

```
schema :
  {
     title: 'SmartContract',
     type: 'object',
     properties:
       {
         contractName : {type : 'string'},
         source : {type : 'textarea'},
       },
     required: ['contractName']
  }
```

Listing 6.6: The Smart Contract artifact instance schema. The only required property is contractName since the user might want to update a smart contract deleting its source code first.

Parser:

```
var parser : Function =
    (model: Object) => {
        //retrieve the list of smart contracts' source codes within
        //the ''Contracts'' directory
        var contracts : string[] = globalThis.retrieveContractNames();

        //get contract name field from form result/model
        //   (it must have this required property)
        var contractName : string = model.contractName;

        //get the ''sourceCode'' field from form result
        //this property could be undefined
                //since the field is not required
        var sourceCode : string | undefined = model.sourceCode;

        //if the user did not specify a source code
                //we assume that **he wants to delete the source code
        if( typeof sourceCode === "undefined" )
                sourceCode = ""

        //contract with name ''contractName'' already exists?

        var contractExists : boolean =
                 contracts.contains( contractName );

        var path = "./Contracts/" + contractName
        files.write(
                path : contractName,
                data: sourceCode,
                // -> Overwrite source code if true
                overwrite: contractExists
                );

        //if contract name exist retrieve its id from globalThis
        //else assign new one associating it to the contract name
        var contractID : number;
```

```
        if( contractExists )
                contractID =
                        globalThis.getContractId( contractName );
        else
                contractID =
                        globalThis.assignContractID( contractName );

        /*return the ID
         the other elements of the triple are "undefined"
           because this parser took care of storing data by itself
           since SCOOP does not foresee the possibility
                                of overwriting files */

        return [
                id : contractID,
                valid_artifact : undefined,
                path : undefined
        ]

    }
```

Listing 6.7: The parser method of the smart contract artifact.

Reviver:

```
var reviver : Function =
  (contractID) =>
    {
        /* Retrieve from the environment the contract name (= file name)
           using the contractID
*/
        var contractName : string =
                globalThis.getContractName( contractID );

        /* Retrieve from the file system the source code of
           the contract file with corresponding name
*/
        var sourceCode : string =
                globalThis.getContractSourceCode( contractName );

        /* Build a model with respect to the
           Smart Contract artifact's schema
           initializing it with the contractName
           and sourceCode just retrieved
```

```
*/
        var contractModel : Object = {
                                            contractName : contractName,
                                            sourceCode : sourceCode
                                        }
        return contractModel
    }
```

Listing 6.8: Smart contracts' reviver. It retrieves from the file system the name and source code of a smart contract given its identifier, previously assigned by the parser method and stored in the global environment object.

Remove:

```
var remove : Function =
  (contractID) =>
    {
        /* Retrieve from the environment the contract name (= file name)
           using the contractID
*/
        var contractName : string =
                globalThis.getContractName( contractID );

        files.delete( path: "./Contracts/" + contractName" )
    globalThis.deleteContract(contractName)
    }
```

Listing 6.9: The remove method for smart contract artifact instances. It simply removes the contract's source code from the file system and the contract name - identifier association within the environment.

Required/Iterable:

```
var required : boolean = false;

var iterable : boolean = true
```

Listing 6.10: It is not mandatory for the user to add smart contracts to the workspace, on the other hand it is possible to add multiple smart contracts.

Network

Schema:

```
schema :
  {
     title: 'Network',
     type: 'object',
     properties: {
        network_name : {type: 'string'},
        host : {type: 'string'},
        port : {type : 'integer'},
        network_id : {type : 'string'},
        /* Following fields are used when the user
           wants to deploy onto a remote blockchain
           entrypoint that **he set up */
        mnemonic : {type : 'string'},
        provider : {type: 'string', enum: ["ropsten", "infura"]}
        httpnode: {type: 'string'}
     },
     required: ['network_name', 'host','port']
  }
```

Listing 6.11: The network artifact instance's schema. It is worth noting that this sub-module supports both local and remote blockchain entry points. In addition, if the user uses a remote node provider, it is possible to authenticate transactions using the "mnemonic" field.

Parser: The correct syntax of the Truffle configuration file expects the "provider" to be a function of the "mnemonic" and the "httpnode". The parser will therefore perform this syntax transformation.

```
var parser : Function =
 (model) =>
    {
        /* Transform the model to the JavaScript object
           with the correct syntax                      */
        var newNetwork : Object = {}
        newNetwork.name = model.networkName
        newNetwork.host = model.host
        newNetwork.port = model.port
        if(model.mnemonic != undefined && model.httpnode != undefined){
           newNetwork.provider =
                 () => {
                    return new HDWalletProvider(
                                          model.mnemonic,
                                          model.httpnode
                                                )
```

```
                    }
        }

        /* Retrieve the contents of the Truffle configuration file,
           which is stored as a JavaScript object and convert it to JSON
*/
        var truffleconfig : Object =
                JSON.parse(files.read("./truffle-config.js"));

        // Retrieve the array of networks within the truffleconfig
        var networks: Array<Object> = truffleconfig.networks

        /* Check if the network name already exists, if that is the case
           store its index within the array
*/
        var oldNetworkIndex : number = -1
        networks.forEach ( (network, index) =>
                                {
                                  if(network.name === model.networkName)
                                        oldNetworkIndex = index
                                }
                         );
        /* If the network does exist,
           remove the old version of the network
        if (networkIndex != -1)
                networks.remove(oldNetworkIndex)

        // Push the new/updated network into the array
        networks.push(newNetwork)

        // Update the Truffle configuration object
        truffleconfig.networks = networks


        // Retrieve the network id from the name or assign a new one
        var networkID : number = getNetworkID( model.networkName )

        return [
                    networkID,
                    // SCOOP subsequently updates the truffle-config file
                    "./truffle-config.js",
                    truffleconfig
                ]
```

```
    }
```

Listing 6.12: The network parser. It simply copies the model properties onto the configuration file, paying attention to "mnemonic" and "httpnode" fields, since their presence implies a more sophisticated handling of "provider" field in the Truffle configuration file.

Reviver:

```
var reviver : Function =
        (networkID) =>
          {
                // Retrieve network name from global environment
                var networkName : string =
                        globalThis.getNetworkNameFromId(networkID)

                // Retrieve truffle's configuration file
                var truffleconfig : Object =
                        JSON.parse( files.read("./truffle-config.js") )

                // Focus on the "networks" property
                var networks : Array<Objects> = truffleconfig.networks

                // Initialize the model that will be returned
                var model : Object = {
                                        network_name : networkName
                                     }

                networks.forEach(
                   (network, index) =>
                        {
                                if( network.name === networkName ){
                                        model.host = network.host
                                        model.port = network.port
                                        model.provider =
                                           parseProvider(network);
                                }
                        }
                );

                return model
          }
```

Listing 6.13: The network's reviver inverts the operations performed by the parser.

Remove:

```
var remover : Function =
        (networkID) =>
          {
                /*First three instructions are the same:
                  Retrieve networkName, truffleconfig,
                        , truffleconfig.networks array
                        [...]                                   */

                var networkIndex : number = -1;
                networks.forEach(
                        (network, index) => {
                                if( network.name === networkName )
                                        networkIndex = index
                        }
                )

                //If the network exists
                if(networkIndex != -1){
                        /* delete it from the array
                                and update the truffle conf. file */
                        networks.delete(networkIndex)
                        truffleconfig.networks = networks
                        files.write("./truffle-config.js", truffleconfig)

                        /*delete the entry that associates
                         the network namewith network id
*/
                        globalThis.deleteNetwork(networkName)
                }

          }
```

Listing 6.14: If the network exists, the remove method uses its identifier to infer the network's name, in order to delete it from the Truffle configuration file.

Required/Iterable:

```
var required : boolean = false;

var iterable : boolean = true
```

Listing 6.15: Adding a network is not required since Truffle's behaviour is to collapse upon a default network if no network is specified. We allow adding multiple networks.

Compiler

Schema:

```
schema :
  {
     title: 'Compiler',
     type: 'object',
     properties: {
        compiler_version: {type: 'string'}
     },
     //if not specified, Truffle uses default compiler version
     required: []
  }
```

Listing 6.16: The compiler's schema is fairly simple. The only property the user may insert is the version of the compiler used to compile each contract.

Parser:

```
var parser : Function =
 (model) =>
    {

        /* Retrieve truffleconfig,
                compiler=truffleconfig.compiler[...] */

        /* If no compiler version is specified
             remove the compiler field in truffle config
               and return an invalid compilerID (=remove)
*/
        if(typeof model.compiler_version === "undefined")
                truffleconfig = truffleconfig.removeProperty(compiler)
        else{
                compiler.compiler_version = model.compiler_version
```

```
                truffleconfig.compiler = compiler
        }




        //in this case compilerVersion is not iterable
        //we therefore omit the ID since there is a unique instance
        return [-1, "./truffle-config.js", truffleconfig]
    }
```

Listing 6.17: The parser removes completely the compiler option in Truffle configuration file if and only if the user sends an empty form, thus manifesting the intention to fall back to the default compiler. The compiler version is a particular case, since only one preference is allowed, we therefore do not need the association between the compiler entity and several identifiers.

Reviver:

```
var reviver : Function =
        (id) => //useless parameter in this case
          {
                /* Retrieve truffleconfig,
                compiler = truffleconfig.compiler  */

                //Initialize the model
                var model : Object = {}

                /* If the compiler field in truffle config is not defined
                   return an empty model, else assign the property */
                if( typeof compiler != "undefined")
                        model.compiler_version = compiler.compiler_version

                return model
          }
```

Listing 6.18: The compiler version's reviver parses the Truffle configuration file looking for the compiler version in order to return a valid intialized model, returning an empty model if the compiler's version is not specified.

Remove: The remove method is semantically equivalent to a "parser" call with an empty model.

Required/Iterable:

```

```

```
var required : boolean = false;

//we do not accept several compiler versions, just the one
var iterable : boolean = false
```

Listing 6.19: The required field is not required, nor it is iterable, since only one compiler version preference is allowed.

6.3.6 Functionalities

To **deploy** and **compile** commands correspond two homonymous functionalities.

Both functionalities will allow the user to subscribe to just one outcome, as to say, the "successful" outcome, which communicates to the user if the functionality is executed successfully or not.

Deploy

As Section 6.1 reports, the migration file is used to automatize the deployment of smart contracts. The deploy command uses this migration script to deploy the contracts. Since our need is to deploy all of the contracts, we need to create a migration script containing all of the contracts added by the user to the Truffle project up until the moment before the deploy command is executed.

First of all, we show how we create such a script in Listing 6.20, following the syntax presented in Listing 6.1.

```
var updateMigrations : Function =
   () => {
            /* Retrieve from the build directory
               the names compiled contracts                    */
            var contracts : Array<string> =
                files.readDir("./Contracts")

            /* We initialize the string that will
               overwrite the content of the migration file */
            var migration : string  = ""

            /* This string will contain the
                several "deployer.deploy(contract_*) lines */
            var deployments : string = ""

            /* The first part of the migration script
```

```
            'requires' the smart contracts' source codes
            so we append to "migration" several lines
            We contextually initialize "deployments"
                                                        */
        contracts.forEach(
            (contractName) => {
                migration.append(
                    "var " +
                    this.removeExtension(contractName) +
                    " = artifacts.require(\"" +
                    contractName +
                    "\"); \n "
                )

                deployments.append(
                    "deployer.deploy(" +
                    this.removeExtension(contractName) +
                    ") \n"
                )
            }
        )

        // Assemble the migration string
        migration.append(
            "module.exports = function(deployer) {" +
            deployments +
            "}"
        }

        // Overwrite the old migration script file
        files.overwrite ( "1_initial_migration.js" , migration)
}
```

Listing 6.20: This function retrieves the names of smart contracts within the Truffle project in order to overwrite the old migration script, updating the contracts that need to be deployed using the correct syntax.

Let us now create the deploy object instantiating parameter and functionality classes described in Figure 4.8.

The functionality name simply is "deploy", the environment object is the global one, thus the "this" object. As anticipated, the only outcome we want to provide is binary and represent whether the deployment is successful. The outcome list we are going to export therefore contains just the "success" string.

Parameters The only parameter we take into account is the "network" parameter. Since Truffle tries a default network whenever this parameter is not specified, we make this parameter optional.

The network parameter is a string, whose domain is a subset of all the possible strings. In our case, the allowed values map out with the network names within the Truffle configuration file, which in turn are provided by the user through the "Network" artifact instance.

The object corresponding to the network parameter is showed in Listing 6.21.

```
var name : string = "network"

var type : Enum<"string", "int"> = "string"

var getValues : Function =
  () =>
    {
        //Retrieve truffleconfig object [...]

        var networks : Array<string> =
           truffleconfig.networks.map(
                (network) =>
                    {
                        return network.name;
                    }
           );
        return networks;

    }

var network : Parameter =
        new Parameter (
                        name,
                        type,
                        getValues
                      );

return network;
```

Listing 6.21: The crucial aspect of the network parameter is to retrieve at runtime the networks stored in the Truffle configuration files through the getValues function.

Body The deploy command's body performs four operations.

- Executes using the shell the Truffle framework command "deploy" passing to it the network string (if present) as parameter.
- Checks if the deployment is successful or not by using the fact that a successful deployment report contains the word "successful".
- Sends the raw output of the command to the log monitor.
- Returns the binary outcome "successful"

Listing 6.22 shows in detail actions performed by the deploy functionality.

```
var deploy : Function =
   (network : string | undefined) => {
        /* Update the migration file to
           include newly added contracts  */
        this.updateMigrations()

        /* If the network parameter is not specified,
        use an empty string                            */
        var networkString : string =
                network === undefined ? "" : network;

        /*Execute the truffle command using the shell
           and store its output                        */
        var output : string =
            shell.execute("truffle deploy " + network)

        /* Check if the output contains the word
           "successful" ;
           otherwise the deployment has failed       */
        var successful : boolean =
             output.includes("successful")

        //send the full report to the log monitor
        this.logmonitor.send ( output )

        //Initialize the outcomes array
        var outcomes : Array<string> = new Array()

        outcomes.push(
                        success: successful
```

```
                    )
        return outcomes
    }
```

Listing 6.22: the deploy functionality simply executes Truffle's deploy command, sending its raw output to the log monitor and returning the global outcome of the operation as a "success" boolean indicator.

Compile

In this adapter, the compile functionality does not accept any parameter, thus the parameter's list is empty. Its name is intuitively "compile" and owns the same environment and outcomes as the deploy functionality. We therefore focus on the functionality's body.

Body The "compile" functionality's body is what we called a "stub method" back in Section 4.3.3, `Q2`. As Listing 6.23 shows, it simply invokes Truffle's "compile" command, retrieving its results without modifying its semantic.

```
var compile : Function =
    () => {
        /*Execute truffle's compile command using
           the shell and store its output           */
        var output : string =
            shell.execute( "truffle compile" )

        /* Check if the output contains the word
           "successful" ;
           otherwise the compilation has failed     */
        var successful : boolean =
             output.includes("successful")

        //send the full report to the log monitor
        this.logmonitor.send ( output )

        //Initialize the outcomes array
        var outcomes : Array<string> = new Array()

        outcomes.push(
                        success: successful
                    )
        return outcomes
    }
```

Listing 6.23: Analogously to what happens in the deployment functionality, this one executes in the shell the compile command of Truffle and retrieves its result basing on the command's outcome

Chapter 7

Conclusions

The main consideration behind our work is that, to the best of our knowledge, it can be challenging for inexperienced users to prototype smart contracts. This means that the current technology for smart contract interaction presents a low degree of discoverability. This is the problem `SCOOP` solves, by creating a layer between inexperienced users and any possible technology that interacts with smart contracts. We therefore manifested the need for understanding the nature of both users and technology. For what concerns the users, we focused on creating a discoverable, user-friendly interface. On the opposite side, the interaction with the underlying technology stack must take into account the numerosity of blockchains, frameworks and smart contract languages. We therefore chose to adopt a modular approach, which is generic towards the different platforms the user might need to interact with. This thesis is divided in two main parts: analysis and implementation.

In the analysis part, we engaged in a thorough study of blockchains, smart contracts, and the current state of the art, coming to understand the need for testing smart contracts and the ways in which current tools achieve it. In particular, it has been of great importance to study smart contract prototyping frameworks, so that we could abstract from them, identifying their common and uncommon aspects. We have also articulated some basic software engineering principles that helped structuring our layer.

The core of our analysis can be found in Chapter 4, where we use the current state of the art as an inspiration for `SCOOP`'s `API`, back-end, front-end and `GUI` design. We ask ourselves questions about common paradigms, features and characteristics a smart contract prototyping tool should present, questions whose answers gave form to our `API`. We also imagined various scenarios the user might encounter. We designed the `SCOOP`'s `API` in such a way that, implementing it, a developer can decide whether to adapt to `SCOOP` an already existing framework/tool (therefore "recycling" it) or to implement

a whole new idea from scratch. Either way, the byproduct is what we call a sub-module. We observed that, by re-using a framework in a sub-module, the sub-module's developer just needs to implement the adapter, without the need of implementing platform-specific behaviour.

This chapter also provides several suggestions about how SCOOP's GUI and workflow should be, highlighting and specifying how the system reacts to users' actions. This has been done providiing several views of the system, both static and dynamic, that aim to be a guide for the implementation.

On the implementation side of our work, we analyze the tools needed to build SCOOP and provide an implementation that exactly reflects the most critical points theorized in the analysis part. In particular, we describe how SCOOP's back-end and front-end perform their main tasks and we implement the API, describing how SCOOP exposes it to sub-modules and how they can, in turn, implement it. In addition, we implement the test script interpreter, i.e. the component that translates the high level instructions coming from the user into the low level functionalities of the sub-module, which in turn may invoke the underlying framework.

In Chapter 6, we create a sub-module from scratch, using Truffle as underlying framework and implementing each component of the API. This module shows that the results found in Chapter 4 can be concretized using the tools and the SCOOP API presented in Chapter 5.

In order to demonstrate the feasibility of both the analysis and implementation's results, we provide in the bibliography the work in progress of SCOOP's prototype [2]. Our prototype uses a restricted version of the API, in order to focus on the most relevant issues.

7.1 Future Work

The most immediately useful step we foresee to take is to implement additional sub-modules, expanding the SCOOP's range of action and improving compatibility with other blockchain platforms, VMs and frameworks.

Once several sub-modules exist, it would be a desirable feature to allow the users to perform quantitative comparison between two or more sub-modules, thus introducing an abstraction to allow different modules to be comparable.

Recording quantitative measures implies adding another level to the SCOOP's infrastructure, which could use a standardised collection of metrics to monitor the amount of resources used by a given functionality within a test script.

SCOOP needs to be user-friendly, in particular we find that the GUI can be improved. Error prompting is limited to the log monitor, although it

would be more helpful for the user to see pop-ups associated with problems occurring in the sub-module.

From an infrastructure perspective, some of the tools we decided to adopt seem to be inadequate for certain tasks. For example, `NodeJS` does not natively handle thread spawning and monitoring, since each separate `NodeJS` environment is single threaded. This feature is desirable when executing workflow scripts, since the adapter's developer may decide to run a tool that idles indefinitely like a server, and the result of executing this sort of script in a `NodeJS` environment would be to block the entire back-end process. This is not particularly limiting since some workarounds to the problem exist, but it is worth evaluating alternatives nevertheless. Last but not least, additional implementation efforts could be spent in bringing the current prototype of `SCOOP`[2] to a fully refined, minimum viable product.

Bibliography

[1] Donald A Norman. *The psychology of everyday things*. Basic books, 1988.

[2] Saverio Catania. *SCOOP's prototype*. In: *GitHub*. Accessed on 21st of September. URL: https://github.com/myricae/SCOOP/tree/master.

[3] Pierre Bourque et al. "The Guide to the Software Engineering Body of Knowledge". In: *IEEE Softw.* 16.6 (1999), pp. 35–44. DOI: 10.1109/52.805471. URL: https://doi.org/10.1109/52.805471.

[4] Timothy Colburn and Gary Shute. "Abstraction in computer science". In: *Minds and Machines* 17.2 (2007), pp. 169–184.

[5] Sungdeok Cha, Richard N Taylor, and Kyochul Kang. *Handbook of software engineering*. Springer, 2019.

[6] *Blockchain*. In: *Wikipedia*. July 14, 2021. URL: https://en.wikipedia.org/wiki/Blockchain.

[7] Karl Wüst and Arthur Gervais. "Do you Need a Blockchain?" In: *Crypto Valley Conference on Blockchain Technology, CVCBT 2018, Zug, Switzerland, June 20-22, 2018*. IEEE, 2018, pp. 1–47. DOI: 10.1109/CVCBT.2018.00011. URL: https://doi.org/10.1109/CVCBT.2018.00011.

[8] David Chaum. "Blind signatures for untraceable payments". In: *Advances in cryptology*. Springer. 1983, pp. 199–203.

[9] L. Lessig G. Gensler. *Smart Contracts and DApps*. In: *MIT Opencourseware*. Accessed on 21st of September. URL: https://ocw.mit.edu/courses/sloan-school-of-management/15-s12-blockchain-and-money-fall-2018/video-lectures/session-6-smart-contracts-and-dapps/.

[10] Usman W Chohan. "The double spending problem and cryptocurrencies". In: *Available at SSRN 3090174* (2017).

[11] Interaction Design Foundation. *What is Prototyping?* In: *interaction-design.org*. Accessed on 21st of September. URL: https://www.interaction-design.org/literature/topics/prototyping.

[12] Sanjeev Sharma. *The DevOps adoption playbook: A guide to adopting DevOps in a multi-speed IT enterprise*. John Wiley & Sons, 2017.

[13] Hans Van Vliet, Hans Van Vliet, and JC Van Vliet. *Software engineering: principles and practice*. Vol. 13. John Wiley & Sons Hoboken, NJ, 2008.

[14] Protocol Labs. *What exactly is Web3?* In: *Web3 Foundation*. Accessed on 21st of September. URL: https://web3.foundation/about/.

[15] Andrew S. Tanenbaum and David Wetherall. *Computer networks, 5th Edition*. Pearson, 2011. ISBN: 0132553171. URL: https://www.worldcat.org/oclc/698581231.

[16] Truffle Suite Developers Team. *Truffle Suite*. In: *Truffle Suite*. Accessed on 21st of September. URL: trufflesuite.com.

[17] Tien Tuan Anh Dinh et al. "BLOCKBENCH: A Framework for Analyzing Private Blockchains". In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by Semih Salihoglu et al. ACM, 2017, pp. 1085–1100. DOI: 10.1145/3035918.3064033. URL: https://doi.org/10.1145/3035918.3064033.

[18] Linux Foundation. *Caliper*. In: *Caliper Documentation*. Accessed on 21st of September. URL: hyperledger.github.io/caliper/v0.4.2/getting-started/.

[19] Dimitri Saingre, Thomas Ledoux, and Jean-Marc Menaud. "BCTMark: a Framework for Benchmarking Blockchain Technologies". In: *17th IEEE/ACS International Conference on Computer Systems and Applications, AICCSA 2020, Antalya, Turkey, November 2-5, 2020*. IEEE, 2020, pp. 1–8. DOI: 10.1109/AICCSA50499.2020.9316536. URL: https://doi.org/10.1109/AICCSA50499.2020.9316536.

[20] Truffle Suite Developers Team. *Truffle*. In: *Truffle Suite*. Accessed on 21st of September. URL: trufflesuite.com/truffle.

[21] Truffle Suite Developers Team. *Ganache*. In: *Truffle Suite*. Accessed on 21st of September. URL: trufflesuite.com/ganache.

[22] Truffle Suite Developers Team. *Drizzle*. In: *Truffle Suite*. Accessed on 21st of September. URL: trufflesuite.com/drizzle.

[23] Truffle Developers Team. *Companies Using Truffle*. In: *Truffle*. Accessed on 21st of September. URL: `trufflesuite.com`.

[24] Electron Community. *Electron documentation*. In: *Electron*. Accessed on 21st of September. URL: `https://www.electronjs.org/docs/latest/`.

[25] Facebook Open Source. *React documentation*. In: *React*. Accessed on 21st of September. URL: `https://it.reactjs.org/docs/getting-started.html`.

[26] Truffle Developers Team. *React JSX Documentation*. In: *Facebook Open Source*. Accessed on 21st of September. URL: `https://reactjs.org/docs/introducing-jsx.html`.

[27] Uniforms Community. *Uniforms Documentation*. In: *Uniforms*. Accessed on 21st of September. URL: `https://uniforms.tools/docs/what-are-uniforms`.

[28] TypeScript Community. *TypeScript Documentation*. In: *TypeScript*. Accessed on 21st of September. URL: `https://www.typescriptlang.org/docs/`.