



## Assignment of master's thesis

<b>Title:</b>	Domain-Specific Languages for Off-chain UI in Decentralized Applications
<b>Student:</b>	Bc. Petr Ančinec
<b>Supervisor:</b>	Ing. Marek Skotnica
<b>Study program:</b>	Informatics
<b>Branch / specialization:</b>	Web and Software Engineering, specialization Software Engineering
<b>Department:</b>	Department of Software Engineering
<b>Validity:</b>	until the end of summer semester 2021/2022

### Instructions

Decentralized applications (dapps) based on blockchain technology are an emerging domain with tremendous potential. However, the dapps can be hard to create and integrate with blockchain infrastructure. This thesis aims to propose a domain-specific language to allow a model-driven solution to dapp UI generation.

Steps to take:

- Review the blockchain dapps, on-chain, off-chain code, blockchain wallets.
- Review standards and approaches to declarative UI development.
- Propose a domain-specific language to define dapps UI compatible with DasContract language.
- Create a proof of concept web application in Blazor.





**FACULTY  
OF INFORMATION  
TECHNOLOGY  
CTU IN PRAGUE**

Master's thesis

# **Domain-Specific Languages for Off-chain UI in Decentralized Applications**

*Bc. Petr Ančinec*

Department of Software Engineering  
Supervisor: Ing. Marek Skotnica

May 1, 2021



---

## **Acknowledgements**

I want to thank my supervisor Ing. Marek Skotnica, for his guidance, mentoring, and consultations. I would also like to thank my family for supporting me during my studies.



---

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 1, 2021

.....

Czech Technical University in Prague

Faculty of Information Technology

© 2021 Petr Ančinec. All rights reserved.

*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

### **Citation of this thesis**

Ančinec, Petr. *Domain-Specific Languages for Off-chain UI in Decentralized Applications*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.



---

# Abstrakt

Tato práce se zabývá definováním a automatickým vytvářením uživatelského rozhraní decentralizovaných aplikací. Hlavním cílem této práce je vytvořit nový doménově specifický jazyk. Tento jazyk by měl umožnit návrh uživatelského rozhraní, které umí komunikovat s chytrými kontrakty.

Nově navržený doménově specifický jazyk je založený na standardních přístupech k deklarativnímu vývoji uživatelských rozhraní zkoumaných v rámci této práce a používá upravenou verzi Extensible Markup Language. Tomuto jazyku by mělo jít jednoduše rozumět bez jakékoliv předchozí znalosti. Jazyk by navíc měl umožnit definování uživatelských rozhraní jak obyčejných chytrých kontraktů, tak chytrých kontraktů z Das Contract projektu. Uživatelská rozhraní vytvořená tímto jazykem jsou implementačně nezávislá a umožňují uživateli komunikovat s chytrými kontrakty nasazenými na libovolném blockchainu.

V rámci této práce byly vytvořeny dvě aplikace na podporu práce s nově navrženým jazykem. Editor formulářů umožňuje návrháři rychle vyvíjet, ověřovat a zobrazovat formuláře vytvořené v novém jazyce. Druhá aplikace používá takto navržené formuláře ke komunikaci s chytrými kontrakty nasazenými na Ethereum blockchainu. Tato aplikace zároveň slouží jako referenční implementace interpretu nově navrženého jazyka. Funkčnost těchto aplikací je ukázána na Das Contract kontraktu pro uzavření decentralizované hypotéky.

**Klíčová slova** blockchain, doménově specifické jazyky, decentralizované aplikace, Das Contract, generování uživatelských rozhraní

# Abstract

This thesis focuses on the definition and generation of decentralized application's user interfaces. The primary goal of this thesis is to create a new domain-specific language that will allow users to design user interfaces. These user interfaces should let the user interact with smart contracts.

The new proposed domain-specific language is based on standard approaches to declarative user interface development researched in this thesis. The new proposed domain-specific language uses Extensible Markup Language modified to be easily understandable without any prior knowledge. It can be used to define user interfaces of both Das Contract and generic smart contracts. User interfaces created with this language are implementation-independent and allow users to interact with smart contracts deployed on any blockchain.

Two proof of concept applications were created to support working with the new domain-specific language. The forms editor allows a designer to develop, validate and render forms in the new language quickly. The forms wallet enables a user to use the forms to interact with smart contracts on the Ethereum network. This application also serves as a reference implementation of the new proposed language. The functionality of both applications is shown on a Das Contract mortgage contract, which allows a user to take out a mortgage without any central authority.

**Keywords** blockchain, domain-specific languages, decentralized applications, Das Contract, user interface generation

---

# Contents

<b>Introduction</b>	<b>1</b>
<b>1 Theoretical foundations</b>	<b>3</b>
1.1 Blockchain technology . . . . .	3
1.2 Blockchain properties . . . . .	4
1.3 Blockchain structure . . . . .	5
1.4 The benefits and dangers of using blockchain technology . . . . .	7
1.5 Ethereum . . . . .	10
1.6 Ethereum wallets . . . . .	10
1.7 Smart contracts . . . . .	10
1.8 Tokens on the blockchain . . . . .	11
1.9 On-chain and off-chain interactions . . . . .	13
1.10 Decentralized applications (DApps) . . . . .	14
1.11 Das Contract . . . . .	16
1.11.1 Mortgage contract . . . . .	18
1.12 Chapter summary . . . . .	20
<b>2 Standards and approaches to declarative user interface development</b>	<b>21</b>
2.1 Three-tier architecture and its relation to decentralized applications . . . . .	21
2.2 Model and view separation and communication approaches . . . . .	22
2.2.1 Model View Controller . . . . .	22
2.2.2 Model View Presenter . . . . .	24
2.2.3 Model View ViewModel . . . . .	24
2.2.4 Summary . . . . .	25
2.3 Declarative vs. procedural user interface . . . . .	25
2.4 Domain-specific languages for declarative user interface . . . . .	27
2.4.1 XML based domain-specific languages . . . . .	28

2.4.2	Domain-specific languages and declarative user interface	28
2.5	Interaction Flow Modeling Language . . . . .	29
2.6	Chapter summary . . . . .	30
<b>3</b>	<b>Towards a new domain-specific language for decentralized application's user interface</b>	<b>31</b>
3.1	First look at the user interface . . . . .	31
3.2	Creating Das Contract forms with Interaction Flow Modeling Language . . . . .	34
3.2.1	Conclusion . . . . .	36
3.3	Creating Das Contract forms with XML-based domain-specific language . . . . .	36
3.3.1	Conclusion . . . . .	39
3.4	Requirements for Das Contract domain-specific language . . . . .	39
3.4.1	Creating forms . . . . .	39
3.4.2	Collecting user input . . . . .	40
3.4.3	Display Das Contract details . . . . .	40
3.4.4	Validate user task roles and control flow . . . . .	40
3.4.5	Basic support for smart contracts outside of the Das Contract methodology . . . . .	40
3.4.6	Easy to extend in the future . . . . .	40
3.5	Redefining Das Contract forms model . . . . .	41
3.6	Domain-specific language requirements evaluation . . . . .	43
3.6.1	Creating forms . . . . .	43
3.6.2	Collecting user input . . . . .	43
3.6.3	Display Das Contract details . . . . .	43
3.6.4	Validate user task roles and control flow . . . . .	45
3.6.5	Easy to extend in the future . . . . .	46
3.7	Forms model as a domain-specific language . . . . .	47
3.8	Chapter summary . . . . .	47
<b>4</b>	<b>Proof of concept</b>	<b>49</b>
4.1	Used technologies . . . . .	49
4.1.1	Blazor WebAssembly . . . . .	49
4.1.2	Nethereum . . . . .	49
4.1.3	Monaco Editor . . . . .	50
4.1.4	Material Design . . . . .	50
4.2	Project scope . . . . .	50
4.3	Use cases . . . . .	53
4.4	Functional and non-functional requirements . . . . .	54
4.4.1	Forms editor . . . . .	54
4.4.2	Forms wallet . . . . .	54
4.5	Architecture and design . . . . .	55
4.5.1	Forms model . . . . .	57

4.5.2	Forms editor . . . . .	58
4.5.3	Forms wallet . . . . .	59
4.6	Development process . . . . .	60
4.6.1	XML serialization . . . . .	60
4.6.2	ViewBind data binding . . . . .	61
4.6.3	ParamBind data binding . . . . .	62
4.7	Testing . . . . .	63
4.8	Project showcase - a decentralized mortgage contract . . . . .	63
<b>Conclusion</b>		<b>67</b>
<b>Bibliography</b>		<b>69</b>
<b>A Acronyms</b>		<b>73</b>
<b>B Contents of enclosed CD</b>		<b>75</b>



---

## List of Figures

1.1	The network view of a blockchain . . . . .	5
1.2	The structure of a generic blockchain block . . . . .	6
1.3	Generic structure of a blockchain . . . . .	7
1.4	Average monthly hashrate breakdown by country . . . . .	9
1.5	A generic oracle data flow . . . . .	13
1.6	A generic model of an oracle and smart contract ecosystem . . . . .	14
1.7	A concept architecture of the Das Contract . . . . .	16
1.8	The Das Contract metamodel . . . . .	17
1.9	The Das Contract mortgage case study . . . . .	19
2.1	Relationship between the three-tier design and MV* patterns . . . . .	23
2.2	Declarative HTML vs. procedural Java . . . . .	26
2.3	Declarative Angular HTML vs. procedural DOM . . . . .	26
3.1	Wireframe of the Das Contract wallet dashboard . . . . .	32
3.2	Wireframe of the Das Contract mortgage contract - apply for a mortgage . . . . .	32
3.3	Wireframe of the Das Contract mortgage contract - pay for a mortgage . . . . .	33
3.4	Wireframe of the Das Contract mortgage contract - cancel a mortgage . . . . .	33
3.5	Subset of IFML elements that could be used for DApps UI . . . . .	35
3.6	IFML diagram of the Das Contract wallet dashboard designed in IFMLEdit . . . . .	36
3.7	UI of the Das Contract wallet dashboard generated from IFML by IFMLEdit . . . . .	37
3.8	Example of syntactic noise in the apply for a mortgage form . . . . .	38
3.9	Removing syntactic noise to increase information value . . . . .	38
3.10	Apply for a mortgage form with minimal syntax noise . . . . .	38
3.11	New Das Contract forms model . . . . .	41
3.12	ViewBind format examples . . . . .	44

3.13	A mapping of the forms model into the XML syntax . . . . .	46
3.14	A form of the apply for a mortgage activity in the mortgage contract	47
4.1	Activity diagram of interaction with smart contract . . . . .	51
4.2	Use case diagram for both applications . . . . .	52
4.3	Package diagram showing essential parts of the project . . . . .	56
4.4	Sequence diagram showing how the forms renderer works . . . . .	57
4.5	Forms model classes with XML annotations . . . . .	61
4.6	ViewToken class used to parse data from the blockchain . . . . .	62
4.7	Final look of the forms editor . . . . .	64
4.8	Final look of the forms wallet while interacting with Das Contract	64
4.9	Final look of the forms wallet while interacting with generic smart contract . . . . .	65
4.10	Final look of the forms wallet while showing Das Contract details .	65



---

# List of Tables

1.1 Proof of Work 51% Attack Cost . . . . . 7



---

# Introduction

A recent surge of interest in blockchain technology has allowed decentralized applications to be more popular. This thesis aims to propose a domain-specific language that will effortlessly define user interfaces that can interact with smart contracts. Most of the effort is put toward defining user interfaces of Das Contract smart contracts, but generic smart contracts ought to be supported as well.

Creating a way to define a decentralized application's user interface quickly has many advantages. It allows developers of decentralized applications to interact with their deployed smart contracts without much effort. It gives users an easy and clean way of concluding smart contract agreements, and together with Das Contract, it should create a new innovative way of developing smart contracts altogether.

This thesis aims to propose a domain-specific language that will allow defining Das Contract and generic smart contract user interfaces in a standardized way and create applications supporting this new language.

The first chapter creates a theoretical basis for blockchain technology and Das Contract, which will be used throughout the thesis. The second chapter focuses on standards and approaches to declarative user interface development. It proposes two approaches to defining user interfaces. In the third chapter, those two approaches are examined and tested and based on the results, a new domain-specific language is proposed. The last chapter deals with designing and implementing the proposed language and two applications supporting it.



---

# Theoretical foundations

This chapter provides an introduction to blockchain technology. It describes how a blockchain works in general, its benefits, and possible issues. Afterward, more focus is put on the Ethereum blockchain and its wallets because the thesis's implementation part will use them. After that, the thesis goes over smart contracts, tokens, decentralized applications, and how on-chain and off-chain code and interactions work. At the end of this chapter, we go over the Das Contract language since we need to propose a domain-specific language that helps define its user interface.

## 1.1 Blockchain technology

Blockchain technology saw its first mention in 2008 in the ground-breaking paper: *Bitcoin: A Peer-to-Peer Electronic Cash System*[1], in which Bitcoin was born, using a chain of blocks (blockchain) as one of its underlying technologies to create a purely peer-to-peer electronic cash that does not need a trusted third party to ensure double-spending does not happen.

**Technical definition:** Blockchain is a peer-to-peer, distributed ledger that is cryptographically-secure, append-only, immutable (extremely hard to change), and updateable only via consensus or agreement among peers.

**Layman's definition:** Blockchain is an ever-growing, secure, shared record-keeping system in which each user of the data holds a copy of the records, which can only be updated if all parties involved in a transaction agree to update.[2]

Blockchain has seen its first use in Bitcoin (peer-to-peer electronic cash). As a result of this invention, other blockchain-based systems have quickly emerged and given the blockchain technology other uses across all parts of the economic sector. Nowadays, blockchain-based systems allow transferring

anything of value securely and without the need for a trusted intermediary. For a regular user, the blockchain provides a platform to send or exchange valuable goods.[3]

A recent surge of interest in blockchain technology has allowed decentralized applications to be more popular. Beyond its financial applications, its potential has come to the foreground in many other sectors, such as trade and supply chains, manufacturing, energy, creative industries, healthcare, and government, public and third sectors. Starting from 2010, the number of new start-ups entering the blockchain industry each year rose steeply worldwide until 2017, at an average annual rate of 40%.[4]

### 1.2 Blockchain properties

If we simplify blockchain, it is almost like a big decentralized database, where every peer is one database node that keeps full duplication of the data. However, as seen in the technical definition, the blockchain has many properties that make it very interesting compared to a regular database. Blockchain is:

**Peer-to-peer** There is no central controller in the network. Participants talk to each other and execute transactions directly. If some of the nodes shut down or face an attack, the system remains functional (certain types of attacks can temporarily put the system at risk, such as 51% attack).

**Distributed ledger** Every peer on the network holds a full copy of the ledger (list of all transactions) same way as a database can be distributed into every node to make sure that data are never lost.

**Cryptographically-secure** Cryptography techniques protect the ledger against tampering. These include data integrity (unauthorized people cannot modify the data), data origin authentication (only a person with the private key can send the transaction), and non-repudiation (a person that has initiated the transaction cannot do anything that would reverse this transaction, assuming the transaction is confirmed).

**Append-only** Transactions can only be added to the blockchain. They are also in chronological order (the transaction can only be added to the blockchain's end). That means once a transaction is in the blockchain, nobody can remove it or alter it by prepending the transaction before it. In rare scenarios, a 51% attack could alter the blockchain.

**Updateable** Any update to the blockchain has to pass the criteria defined in the blockchain protocol and a consensus, meaning all the peers have to agree to this change. There are many different consensus algorithms. The most prominent blockchains use Proof of Work (voting by computation power, also known as mining) or Proof of Stake (voting based on the stake - the amount invested into the system).[2]

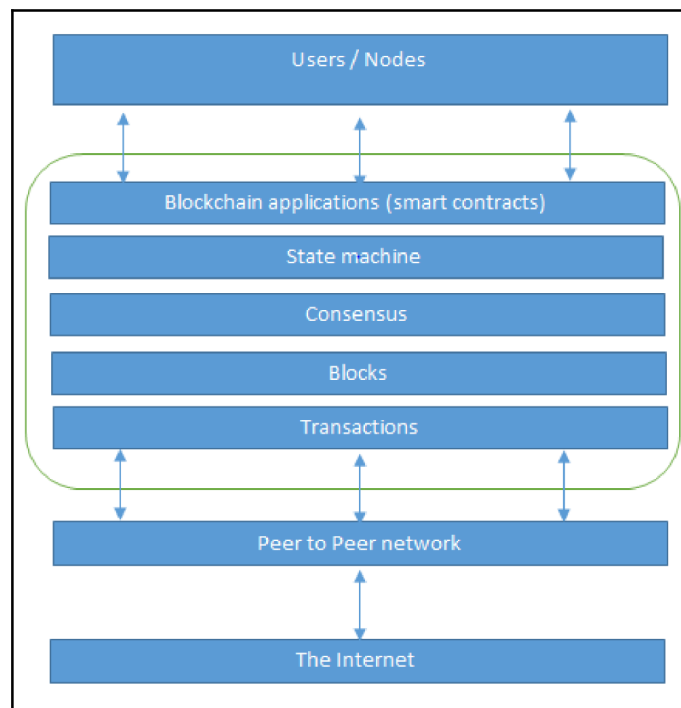


Figure 1.1: The network view of a blockchain[2]

### 1.3 Blockchain structure

Blockchain can be imagined as a layer of a peer-to-peer network that uses the internet for communication; it is similar to HTTP running on top of TCP/IP. Figure 1.1 shows the network view of the blockchain. The peer-to-peer network hosts the blockchain system that contains transactions, blocks, consensus mechanisms, state machines, and smart contracts. Finally, the users connect to the blockchain and perform tasks like executing transactions, verifying transactions, and voting.[2]

The blockchain consists of blocks chained together. Each block, as seen in figure 1.2, consists of a header and a body. The body consists of many transactions bundled together. A single transaction is a record of some event that occurred, for example, a user sending money to another user. The header contains information required to keep the blockchain valid. The blocks are chained together using the previous block's hash. The only exception is the first block, which is hardcoded into the chain when it runs for the first time. A nonce is a number that the miners need to find in the Proof of Work consensus. When this number is found, the block is mined, and the miner receives a reward. The Merkle root is a hash of all nodes in the Merkle tree; this allows all transactions within the block to be verified at once, speeding up

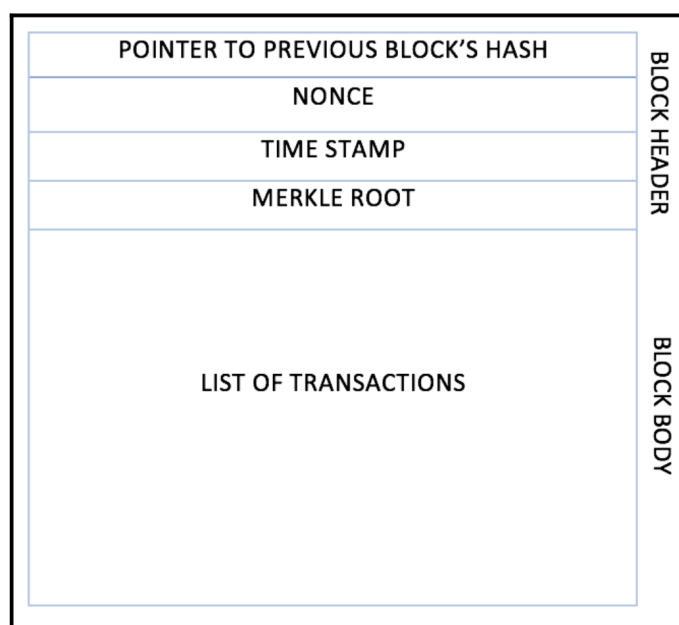


Figure 1.2: The structure of a generic blockchain block[2]

the verification process. The timestamp is used to calculate the time between two blocks being mined; this allows the Proof of Work-based blockchain to adjust mining difficulty according to the mining power.[2]

The blockchain consists of blocks chained together. Each block, as seen in figure 1.2, consists of a header and a body. The body consists of many transactions bundled together. A single transaction is a record of some event that occurred, for example, a user sending money to another user. The header contains information required to keep the blockchain valid. A nonce is a number that the miners need to find in the Proof of Work consensus. When this number is found, the block is mined, and the miner receives a reward. The Merkle root is a hash of all nodes in the Merkle tree; this allows all transactions within the block to be verified at once, speeding up the verification process. The timestamp is used to calculate the time between two blocks being mined; this allows the Proof of Work-based blockchain to adjust mining difficulty according to the mining power.[2]

When a node needs to transfer something over a blockchain, it creates a transaction and signs it with its private key. The transaction contains a source and a destination address, plus other validation information. It can also contain business logic, such as rules that need to be fulfilled or how the value is transferred. The transaction is then propagated into the network using a Gossip protocol to transaction validators. When multiple validators validate the transaction, it is included in the next block, then propagated into



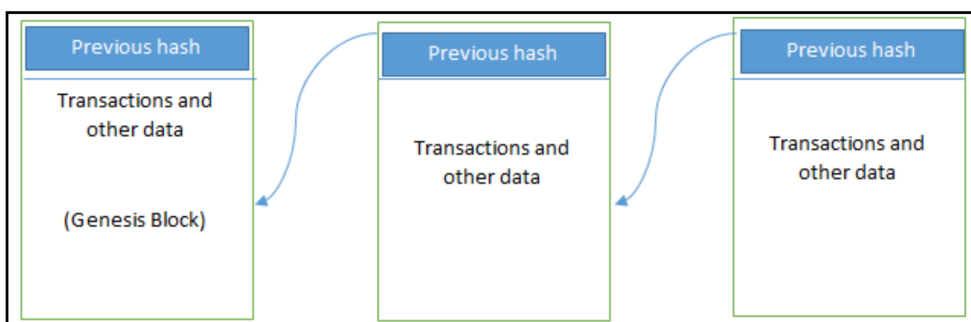


Figure 1.3: Generic structure of a blockchain[2]

the network. The blocks are chained together using the previous block’s hash, as seen in figure 1.3. The only exception is the first block, which is hardcoded into the chain when it runs for the first time. Each time a new block is added, all previous blocks receive one confirmation. When enough blocks are appended after the block with the node’s transaction (in Bitcoin, this is usually six blocks), the transaction is considered final and irreversible.[2][3]

## 1.4 The benefits and dangers of using blockchain technology

The following section goes over some of the most notable benefits and drawbacks of using blockchain technology. Since blockchain technology is still relatively young, many challenges and problems arise, and it is crucial to keep them in mind. As of right now, there are many different types of blockchain, each one focusing on a different spectrum of problems. So while many of these problems have been solved already, we still do not have a blockchain technology, which would deal with most of the problems, same as we do not have a universal programming language. Some of the most important advantages are:

Table 1.1: Proof of Work 51% Attack Cost[5]

Name	Market cap	Hash rate	1h attack cost	NiceHash
Bitcoin	\$355.17 B	114,915 PH/s	\$716,072	0%
Ethereum	\$67.76 B	253 TH/s	\$418,438	3%
Litecoin	\$5.81 B	227 TH/s	\$29,287	6%
BitcoinCash	\$5.44 B	1,374 PH/s	\$8,560	33%
Dash	\$1.04 B	7 PH/s	\$3,246	2%

**Transparency and trust** Because changes to the ledger are visible to everyone on the network, and the transactions cannot be altered or removed, the system is fully transparent, and it is nearly impossible to commit fraud by modifying the data.

**Decentralization** Removing the need for an intermediary further increases trust in the system, as there is no central authority one would need to trust not to commit fraud (in countries with significant corruption, trusting central authority is a big problem). Another advantage of not having an intermediary is the speedup of the transaction process.

**Security** Modifying or removing data in the blockchain is nearly impossible. In the past, there have been many cases of fraud by manipulating the data. In the blockchain, every transaction can be traced by anyone on the network. Regular systems are prone to hacking because they can be very complex and intertwined. While a 51% attack could alter the blockchain, it is nearly impossible to execute 51% attacks on large blockchains. Additionally, it could be quickly spotted by other nodes in the network.

**Cost-saving** Having an intermediary usually involves additional fees when transferring assets. There can be multiple intermediaries, each requiring a fee to provide a service, maintain their ledger, and exchange data.

**High-availability** Blockchain is based on a peer-to-peer network with thousands of nodes, each keeping its copy of the ledger. Even if many nodes leave the network or are taken down, the blockchain will remain accessible. This extreme redundancy ensures the high availability of the blockchain.[2][3]

Problems and challenges the blockchain technology needs to face are:

**Lack of privacy** Since everything on the blockchain is visible for everyone, it is relatively easy to figure out an account's identity after payment is received from them. While this problem is being worked on and some blockchains already offer nearly untrackable transactions, major blockchains still do not have a solution implemented to protect the regular uninformed user.

**Lack of central control** While having no central control brings advantages to the blockchain by giving control back to users, it also gives the users responsibility to educate themselves and care for their assets. The user needs to be aware of various scams and responsibilities, such as keeping their private key safe, making sure he does not lose it, and at the same time does not show it to anyone. The blockchain can be very unforgiving; if a user gets scammed or sends assets to the wrong address, it will most likely be gone forever.

## 1.4. The benefits and dangers of using blockchain technology

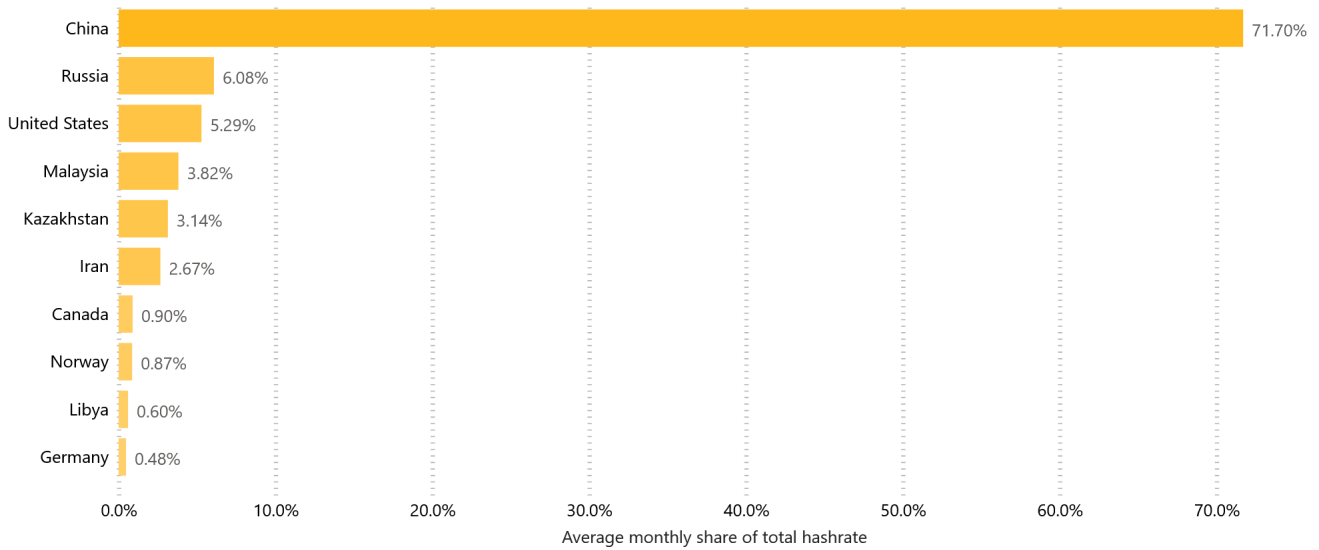


Figure 1.4: Average monthly hashrate breakdown by country (from September 2019 to April 2020)[6]

**51% attack** If a blockchain uses Proof of Work consensus, if an attacker were to obtain more than 50% of computing power, he can control the blockchain's transactions, known as a 51% attack. Table 1.1 shows how much would a 51% attack cost for an hour and what percentage of this power could be borrowed from NiceHash (one of the most significant rentable mining services). While it might look like big blockchains cannot be vulnerable to this attack because of the price and coordination required to pull it off, if we look at graph 1.4, we can see that China owned roughly 72% of all Bitcoin's hash rate on average from September 2019 to April 2020.

**Scalability and cost** Proof of Work requires proof that computing power and resources were contributed to the network before a block is added. Enormous amounts of electricity and computing power are required to run the network, essentially wasting it on computing hashes. When there is an exponential increase in transactions on the network, the cost per transaction increases because it is not ready to handle this amount of transactions. For example, a block is added into the Bitcoin blockchain every 10 minutes and contains around 2000 transactions so the Bitcoin network can handle approximately three transactions per second.[3]

## 1.5 Ethereum

Ethereum is an open-source, globally decentralized computing infrastructure that executes smart contracts. From another view, Ethereum is a deterministic state machine with a globally accessible singleton state and a Turing-complete virtual machine (EVM) that changes this state by executing code. Blockchain technology allows keeping the history of state changes. Ethereum tracks the state transitions of a general-purpose data store (anything that a key-value pair can represent). It can store both code and data, meaning it can perform more than just monetary transactions.[7]

The Ethereum Virtual Machine (EVM) can load code into its state machine and run it, storing its blockchain's resulting state changes. Since it is Turing-complete, it can compute any algorithm that any Turing machine can compute. Running code in the EVM is very expensive because the code runs on many nodes. It also introduces the halting problem, which proves that we cannot determine if a program will stop running unless we run it. To ensure there is not a smart contract that would run an infinite loop, halting the EVM or just simply wasting resources, the EVM charges a fee for every instruction executed on it. Every instruction has a predetermined cost in gas, which has to be bought with ether (Ethereum's main currency unit) to keep the code running and reward nodes executing it.[7][8]

## 1.6 Ethereum wallets

In Ethereum, wallets can have multiple functionalities. A wallet is a software application that serves as a primary user interface to Ethereum. Every wallet allows its user to manage his keys, addresses, and balance. It also enables him to create and sign transactions. On top of that, some Ethereum wallets allow the user to interact with smart contracts. Therefore, they can serve as an interface for decentralized applications. Wallets that allow DApp interactions are usually either a browser extension or mobile app, allowing seamless integration with websites providing DApps.[7]

MetaMask[9] is an example of an Ethereum wallet that also serves as a gateway to DApps. It allows users to store and send ethers and tokens. Users can then connect the wallet to Ethereum-based DApps and spend the ether or tokens in games, gambling, or DeFi exchanges. Once the DApp connects to the wallet, it can view the wallet address and ask the user for payment through MetaMask.[7][10]

## 1.7 Smart contracts

Nick Szabo first defined smart contracts in an article called *Formalizing and Securing Relationships on Public Networks*[11] in the 1990s:

A smart contract is an electronic transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs.[11][12]

A typical example of automatic execution is a vending machine. It has an internal state and behaves algorithmically, meaning the same instructions are executed every time. Once a person inserts money and selects the goods, they are released. The machine cannot decide not to follow the contract. Similarly, a smart contract has to execute the programmed code.[13]

Blockchain technology plays a vital role in providing the necessary underlying network with security guarantees required to run the smart contracts.[12] We will now focus only on Ethereum smart contracts, where smart contracts usually refer to immutable computer programs that run on the EVM as part of the Ethereum network protocol. The smart contract is:

**Autonomous and immutable** After the contract is launched and running, nobody can stop it from automatically executing the agreement, and the code of the smart contract cannot change unless a new instance is deployed.[13][7]

**Self-sufficient** The contract can automatically gather required resources to run itself (storage, processing power) by issuing equity, raising funds, or providing services.[13]

**Deterministic** The outcome of the smart contract's execution is the same for every node that runs its code in the context of the transaction that initiated it. Since executing smart contracts results in a change of state of the Ethereum blockchain, all nodes need to agree on it using a consensus mechanism. Therefore, it is required that there is no random behavior and the execution is deterministic. If it were not, the nodes would never agree on a state of the Ethereum blockchain.[7]

**Decentralized** The contract does not reside on a single node but is distributed and executing across network nodes in the EVM. Thanks to the deterministic behavior, all instances of the EVM achieve the same result, and therefore it seems like the system operates as a single global computer.[13][7]

## 1.8 Tokens on the blockchain

A blockchain token is a coin-like item with a specific purpose. Tokens can be programmed to serve many different functions. The most common use for

a token is it being a currency. Other functionalities include asset ownership (token representing an asset such as gold, oil, or energy), access (token representing access rights, for example, a right to use a service), equity (token representing shareholder equity), voting (token representing voting rights in a legal system), identity (digital ID) and many others. Tokens can be fungible and non-fungible.[7]

Fungible tokens are not unique, meaning we can substitute one token for another of the same kind, and the value and function stay the same. Fungible tokens can also be divided into smaller parts. The most obvious case of this is currency. The value and functionality of one ether stay the same as any other ether. Further, we can divide one ether into smaller units (gwei), and they will be indistinguishable from others.[7][8]

On the other hand, non-fungible tokens each represent a unique item, and one is not like the other. For example, a token representing a specific person's digital identity cannot be replaced with the same token type, depicting another person's digital identity. Therefore, non-fungible tokens need to have a unique identifier, and they cannot be split into smaller parts.[7][8]

On the Ethereum platform, tokens are different from the ether because the Ethereum protocol doesn't know anything about them. Ether transactions are an action of the Ethereum platform, but sending and owning tokens is done only on the smart contract level. Every token needs to have an underlying smart contract handling the operations possible with this token. Once the token is deployed, the smart contract manages everything and needs to be programmed without errors. While a developer can write a token from scratch, it is wiser to use existing standards to avoid creating unnecessary code bugs.[7][8]

The first Ethereum token standard is called ERC-20. It is the most used standard to date, and it is used for fungible tokens, meaning ERC-20 tokens are interchangeable and have no unique properties. The ERC-20 standard defines a common interface, which allows tracking balance for every token holder address and provides the ability to manage this balance (send, receive). The ERC-20 tokens create a simple way to create a cryptocurrency for a project on top of the Ethereum network.[14][7]

For non-fungible tokens, there is the ERC-721 standard. It is inspired by the ERC-20 standard, except it is changed to represent non-fungible tokens, making it easy to create tokens representing unique items such as collectibles or ownership. Since ERC-721 tokens are unique, the user doesn't own tokens just as a number but instead holds a particular set of identifiable tokens, each one representing something else.[15][8]

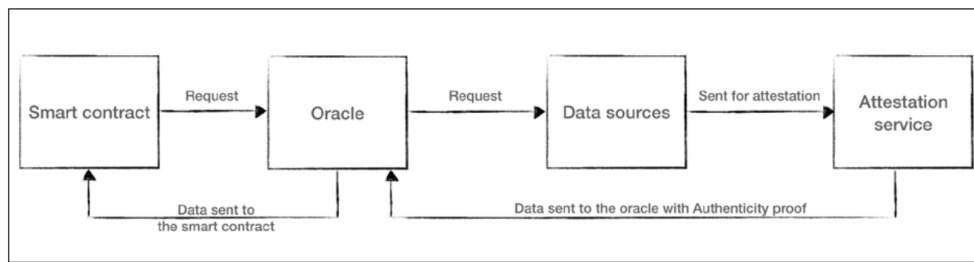


Figure 1.5: A generic oracle data flow[12]

## 1.9 On-chain and off-chain interactions

Blockchain is a closed ecosystem, meaning it cannot naturally interact with the real world outside its network. When a smart contract is deployed and running, it executes on-chain code. On-chain code is typically small and contains only the smart contract’s crucial parts as executing it is costly. Off-chain code does not run on the blockchain and introduces additional required trust by the users, as it does not have the same properties as on-chain code. We would want to eliminate any off-chain code in an ideal scenario as it undermines the smart contract’s trust. However, it is not possible because the blockchain does not have access to the real-world, which is usually required for the smart contract to be useful. Additionally, off-chain code allows us to reduce the amount of code running on-chain, which drastically reduces the costs of running the smart contract. Oracles allow us to address these issues.[12][7]

Oracles are trusted entities that use a secure channel to transfer off-chain data to a smart contract to execute its business logic. For example, a smart contract might need to access prices of stocks, cryptocurrency exchange rates, random numbers, scores of matches, or if the other party has received property or goods. They can also be used to relay data securely to the frontend of the DApp. The steps of how an oracle works are shown in figure 1.5 and below:[12][7]

1. A smart contract requests data from an oracle.
2. Oracle executes the request and retrieves the requested data from the source.
3. The data are sent to a notary where they are digitally signed to prove their authenticity. There are other techniques to provide trusted data as well.
4. The signed data are sent back to the oracle.

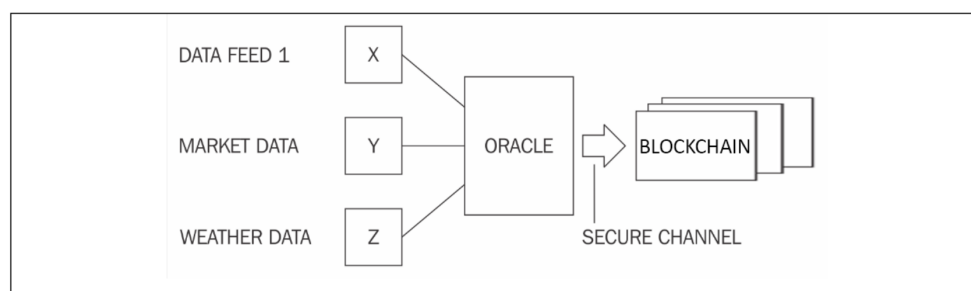


Figure 1.6: A generic model of an oracle and smart contract ecosystem[12]

5. Optionally, the signed data can be stored on a decentralized storage system like Swarm if they are too big for the smart contract.
6. Finally, the signed data are sent to the smart contract.

While centralized oracles are good enough for many applications, they present a security threat to the Ethereum network. For example, let's take a cryptocurrency exchange rate from a single centralized exchange and use it to create a smart contract for decentralized exchange. There is no guarantee that the service will have 100% availability at all times. Additionally, should the exchange choose to or be subject to an attack, they could manipulate the exchange rates sent to the smart contract for their own gain. Decentralized oracles are trying to solve these issues by providing tamper-resistant data.[7][16]

ChainLink has proposed a decentralized oracle network that serves as middleware between smart contracts and external data sources. It consists of three smart contracts: a reputation contract, an order-matching contract, and an aggregation contract. The reputation contract is meant to keep track of data providers' performance. The order matching contract is responsible for selecting bids from oracles using the reputation contract and creating a service-level agreement based on the number of oracles required. The aggregation contract then collects responses from multiple oracles, calculates the final result, and feeds it back into the reputation contract.[7][16]

Instead of having a single data feed from a centralized exchange, there would now be an oracle providing the smart contract with aggregated data from multiple exchanges, based on their reputation and previous performance.

## 1.10 Decentralized applications (DApps)

A decentralized application (DApp) is an application that does not run on a central server (or cluster). A typical application consists of frontend, back-



end, and data store. On top of that, it might also communicate with other applications or systems, and if a user wants to connect, he might need a name resolution service. All of these parts can be somewhat centralized and somewhat decentralized.[12]

In an ideal scenario, a DApp should be entirely open-source and autonomous, meaning it can raise funds on its own to cover the costs of running it. It would use a smart contract as a backend, with the data and records of operations being stored on a public, decentralized blockchain or P2P storage. Any changes to the DApp should be done via consensus, with nobody having majority control over the application. The application's front end should be an app that runs directly on a device rather than a centralized server.[12][7]

Most of the DApps are only partially decentralized while still using centralized services where they are needed or convenient. Running code and storing data on the Ethereum platform is expensive because of its current proof of work consensus and high ether costs. That is why the smart contract is used only for the core features, while everything else is computed off-chain.[7]

DApps have advantages that a typical centralized application cannot provide:

**Resiliency** Since a smart contract controls the business logic, a DApp is distributed across the blockchain platform, it cannot be altered, and its data and transactions are stored persistently. It will also have no downtime as long as the platform is still operating.

**Transparency** Anyone can inspect the code of a smart contract on a blockchain to ensure that it does what it is intended to do. All interactions with the DApp are stored forever in the blockchain.

**Censorship resistance** As long as a user can access a blockchain node (or run one), he can always interact with the DApp without interference.[7]

Generally, DApps provide different services. The main ones are finance, gaming, gambling, social media, transportation, and shopping. Here are some examples:

**DeFi** stands for Decentralized Finance, and it is one of the most attractive concepts for DApps. Traditionally, finance is a business that was almost impossible to do without involvement from a third party and a fee for their services. Users could be using decentralized exchanges to buy cryptocurrencies, stocks, take loans, or get insured without a central authority that needs to be trusted.[12]

**OpenBazaar** is a decentralized peer-to-peer network that allows sellers and buyers to interact directly without a third party. While the network does not run on a blockchain, it uses cryptocurrencies for payment.[12][13][17]

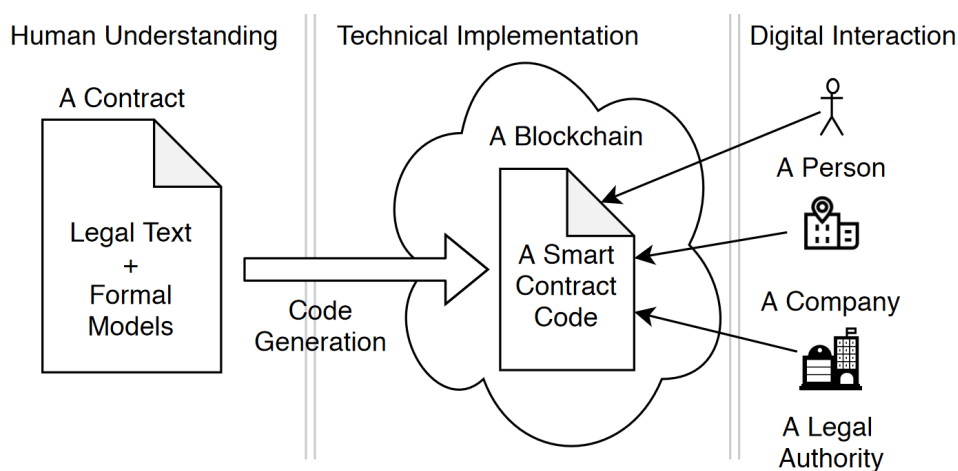


Figure 1.7: A concept architecture of the Das Contract[25]

**LaZooz** is a decentralized version of Uber, allowing users to rideshare. There are many similar projects, such as Eva and Drife.[18][19][20]

**Twister** is a social network similar to Twitter or Facebook, allowing its users to microblog.[13][21]

**MakerDAO** is a decentralized lending platform built on Ethereum to allow lending and borrowing of cryptocurrencies without the need for a middle man using smart contracts.[22]

**CryptoKitties** is a game built around breeding and collecting virtual cat pets.[23]

**Decentraland** is a virtual game world where users can buy land and vote using DAO tokens.[24]

## 1.11 Das Contract

The Das Contract is a Domain Specific Language (DSL) for specifying blockchain smart contracts. It uses a subset of BPMN 2.0 and other concepts specific to blockchain technology to create platform-independent models. Using model-driven engineering allows automatic generation of smart contracts for supported platforms from these models. According to the *Das Contract paper* [25], its main focus is to deal with law and legal contract ambiguity, and its approach to resolving contract ambiguity consists of three parts, as seen in figure 1.7:

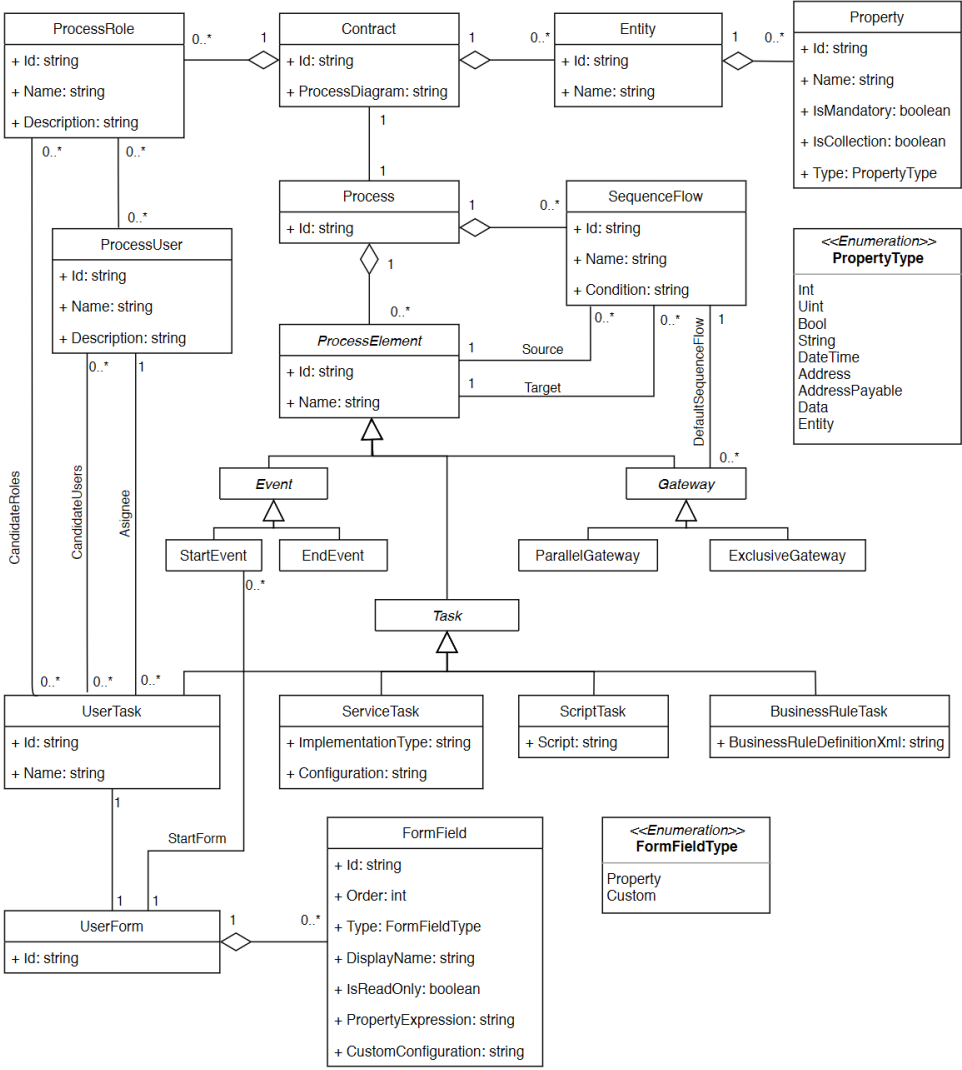


Figure 1.8: The Das Contract metamodel[26]

**Human Understanding** part defines a contract between multiple parties that they need to agree on. Such a contract is a combination of legal text and formal ontological models. The legal text in some form specifies the legal validity of the formal model. The formal models need to be unambiguous, so only one possible interpretation is allowed.

**Technical Implementation** part specifies how formal models from the contract are transformed into a software executable code and uploaded into a blockchain as a smart contract.

**Digital Interaction** is a part where people, companies and legal authorities can interact with the agreed upon contracts. Since the contract is in a blockchain, the interaction is fully digital, and thanks to cryptography can also be legally binding. Blockchain by design also provides an audit trail of all actions performed by the parties and ensures that the agreed upon contract is executed correctly.[25]

The Das Contract consists of three interconnected models, as seen in figure 1.8:

**The process model** is based on an extended subset of BPMN 2.0. It specifies the smart contract's process activities, execution order, user roles, and property mappings. It also supports working with both fungible and non-fungible tokens.

**The data model** is based on a UML class diagram. It is used to specify entities, properties, and relations between them. An entity may contain primitive data types as well as addresses for token support. In the model, tokens are represented as a special entity using inheritance.

**The forms model** specifies an interface for the user input. It is then generated into an on-chain code, which handles validations, user rights, property bindings, and an off-chain model, which is interpreted by a blockchain wallet that allows the user to interact with the smart contract. The on-chain code is equivalent to server-side code, and the off-chain code is equivalent to client-side code in a regular application.[28]

### 1.11.1 Mortgage contract

In the following chapters, we will be using the Das Contract mortgage contract[27] as a practical example to show different approaches to defining its UI. In figure 1.9, we can see the Das Contract diagram of the mortgage contract. It depicts four roles: borrower, lender, insurer, and property owner, each having different responsibilities and actions based on their roles. Some

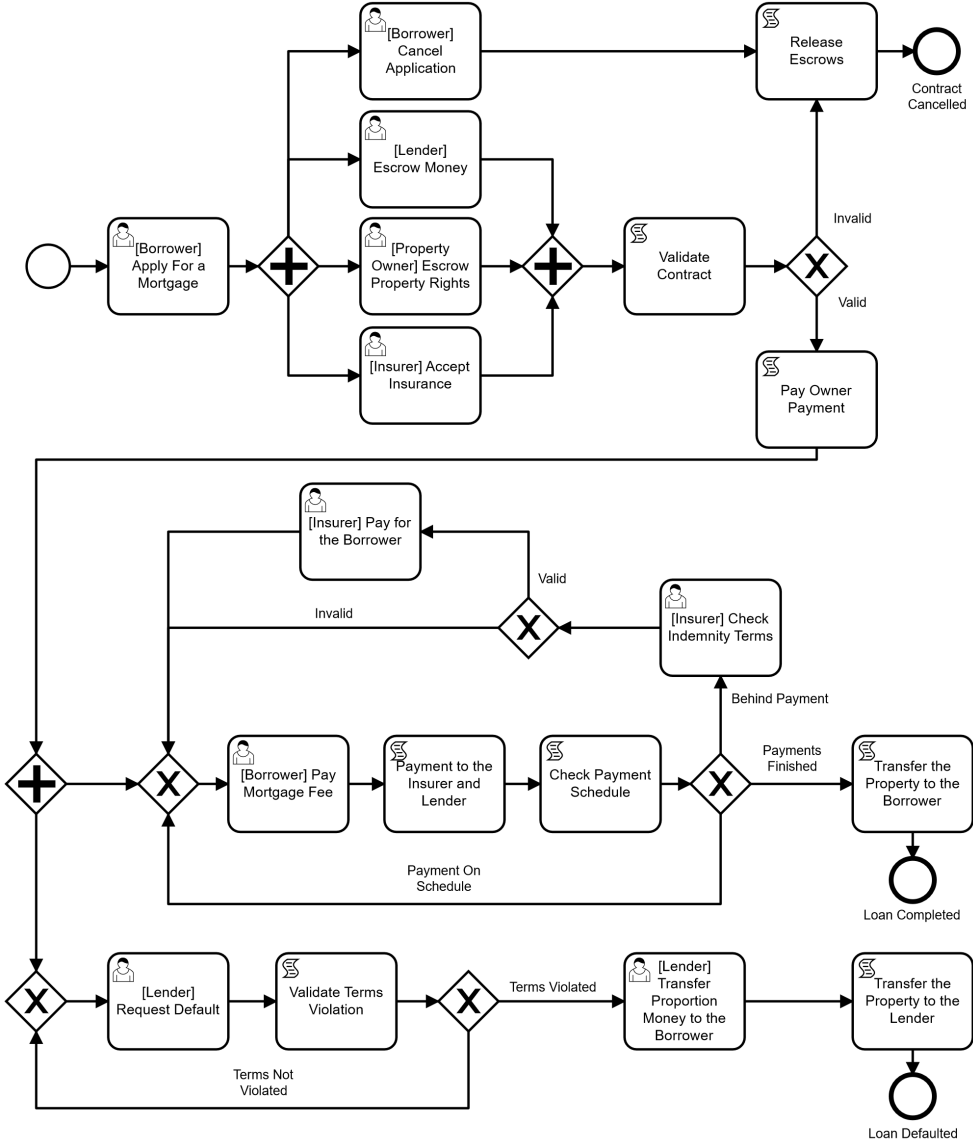


Figure 1.9: The Das Contract mortgage case study[27]

of these actions have a form attached to them. For example, the borrower can apply for a mortgage, pay a mortgage fee and cancel a mortgage application using the forms from the Das Contract forms model. The mortgage case diagram has been generated into a Solidity smart contract and can run on the Ethereum blockchain.

### 1.12 Chapter summary

This chapter has gone over how blockchains work, their main advantages and disadvantages, and how we can use them to transfer value and execute contracts automatically. We have also shown different types of tokens, which can represent valuable assets or voting rights, and wallets, which are used as a gateway to decentralized applications. Near the end, we introduced the idea of fully decentralized applications and how we can use oracles to deliver off-chain information on-chain. Finally, the Das Contract language, which can generate smart contracts from platform-independent models, was reviewed to understand it better and propose a domain-specific language for generating its user interface.

---

# Standards and approaches to declarative user interface development

This chapter describes standards and approaches to defining user interfaces declaratively. In the first part, the three-tier architecture is quickly revised, and a possible mapping to decentralized applications is shown. After that, the MV\* pattern family is discussed, its relation to decentralized applications and three-tier architecture is shown. Afterward, domain-specific languages are reviewed, showing their advantages and how we can use them to declare user interfaces. In the end, the Interaction Flow Modeling Language is examined to present another possible way of declaring user interfaces as a standardized platform-independent model.

## 2.1 Three-tier architecture and its relation to decentralized applications

Enterprise applications can be divided into three essential layers:

**The presentation layer** handles the interaction between the user and the software. Its primary responsibility is to display information to the user and interpret the user's commands into actions.

**The business layer**, also known as business logic, represents the core functionality of the domain. It works with both inputs and stored data, validates them, enforces business rules on them, and uses them to provide the user service beneficial to him.

**The data access layer** is responsible for retrieving and storing data either from a database or other systems.[29][30]

These three layers can also be mapped to DApps. The more centralized a DApp is, the easier it is to see. The presentation layer is usually the same as in enterprise applications, whether it is a website or a standalone application. The business logic consists of one or many smart contracts, and the data access layer is represented by oracles and other storages, such as P2P storage or blockchain data storage.

Currently, many DApps are still relatively centralized. We can look at them as enterprise applications that use smart contracts only for a particular purpose, such as storing the essential data on a blockchain to maintain transparency and provability. Therefore, when creating a DSL for DApp UI, common patterns that allow separation of concerns should still be used so that the UI can be exchanged for another one when needed.

## 2.2 Model and view separation and communication approaches

Separating the model from the view is the first phase in developing larger applications. Some of the most significant responsibilities - working with data and presenting data - are now fenced off. However, this is usually not sufficient. The view is not insulated from the model, and they rely on each other. A mediator is required to provide communication between these two components so they can be replaced when needed. The patterns from the MV\* family are well established and provide different approaches for separating the model and the view of an application.[29][30]

### 2.2.1 Model View Controller

MVC is one of the oldest and most influential patterns that tried to solve the view and model separation in response to a need for large-scale systems with UI. In MVC, a model is an object that represents information about the domain and state of the application. It contains all the data and behavior that is not related to UI, is entirely ignorant of the UI, and can function independently.[30][31]

The view has a single functionality of getting data from the model and displaying it. The controller handles user input and manipulates the model. Therefore, the MVC's UI consists of the view and the controller, and the model is not aware of them. When mapping the MVC components to the three-tier architecture, the model and controller overlap across multiple tiers, as shown in figure 2.1.[29][30]

One of the MVC pattern's main issues is that large systems that use multiple view-controller pairs for a single model need a way of synchronizing the view when the model changes. Since there is no data binding in the MVC pattern, all view-controller pairs observe the model for changes. There is also a



2.2. Model and view separation and communication approaches

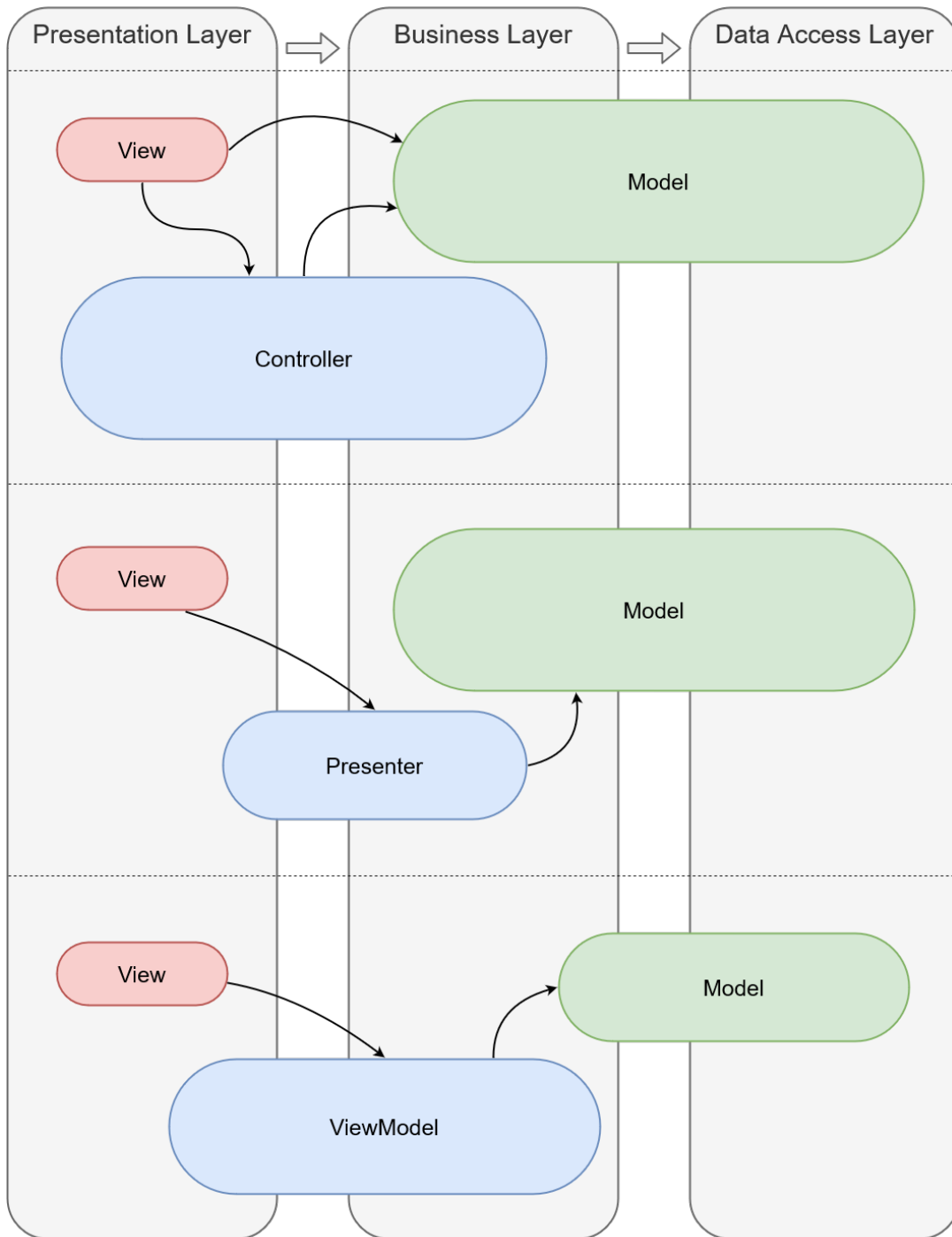


Figure 2.1: Relationship between the three-tier design and MV\* patterns[29]

potential for excessive updates if many views exist on the same model. Since view-controller pairs make direct calls to the model, changes to the model interface break both view and controller code. Creating a new view also requires a new controller because they are closely coupled and usually cannot be reused.[30][32]

The MVC pattern has several benefits. It is easy to create multiple views running concurrently. Therefore, there can be both mobile and desktop applications running on the same model. There can also be multiple views within a single application (for example, a view for a new user and a view for an experienced user). Since the view-controller pairs observe the model for changes, all views are always synchronized. The separation of the view and the model allows developers who program the model to focus on business logic, and they do not have to consider how the UI will be done. The UI part of the application can use different technology and platform, and it is relatively easy to switch when the UI looks outdated.[30][32]

### 2.2.2 Model View Presenter

The MVP pattern is trying to address the shortcomings of the MVC. The model remains the same. The controller is replaced by the presenter, which still handles the user input and manipulates the model, but additionally, it updates and synchronizes the view when needed. Therefore, the view does not need to be aware of the model and can be tested independently. The presenter can also store the state of the view, and it performs more complex business logic; it also communicates with the view through an interface, so it does not rely on implementation details. Overall, the three components have looser coupling and more flexibility than in the MVC.[29][33]

### 2.2.3 Model View ViewModel

The MVVM pattern is an alternative to the MVC and MVP patterns. The main reason for its creation was that the view's state was still linked to the model in both MVC and MVP patterns, which makes testing of individual components harder and interferes with the principle of modular programming.[29][34]

In MVVM, the view model replaces the presenter/controller, and its responsibilities change again. The model is still used to access data sources, but it does not perform too much business logic, and it is not aware of the view or the view model. The view typically uses very limited code-behind (usually, it is only defined by a markup language), presents structured data to the user in its current state, and collects user interaction and events. The view can then forward events and trigger actions on the view model, and it also updates its properties through two-way or one-way data binding (the data that change in the view are automatically changed in the view model and vice versa). The

view model is not aware of the view, which makes testing and substitution easier. It is responsible for passing data from the view to the model in a format the model can understand, and it makes the model data available to the view through data binding. The view model is also responsible for view logic, state changes, and data validation.[29][34]

The main advantage of the MVVM pattern is the complete separation of the view and the view model. The view, view model, and model can all be developed concurrently and tested independently. It is also easier to redesign the UI when needed.[29][34]

### 2.2.4 Summary

There are three main approaches for separating the view from the model. When fitting these patterns to the decentralized application approach, the MVVM pattern is the best candidate. Let's say that the smart contract of a decentralized application is the model in the MV\* pattern family. The MVP and MVC approach might seem more fitting because in a truly decentralized application, the smart contract should do all business logic and data-related operations, and the off-chain code should consist of a simple view of the smart contract. However, the current fees for running code on the blockchain (for example, on the EVM) do not allow for truly decentralized applications. Instead, the smart contract should only contain the core functionality, and the remaining business logic has to be done off-chain. For this approach, the MVVM pattern is the most fitting. The model has fewer responsibilities; the view model can serve for business logic and data validation that does not have to be done on-chain, and it also handles the view's state and logic. Another advantage of the MVVM pattern is that it is very easy to couple with declarative UI, which allows us to create a simple declarative DSL for UI in DApps.

## 2.3 Declarative vs. procedural user interface

The main difference between declarative and procedural UI and programming, in general, is the approach. In procedural programming, we are describing how to achieve a goal. In declarative programming, we describe what we want, and the interpreter has to figure out how to achieve it for us. Declarative UI also provides a higher level of abstraction. The prime example of declarative UI is HTML. In HTML, we declaratively specify how the view should look and what elements should be placed where. It is up to the web browser to parse the HTML elements and create a view for us. To modify the view further, we can also use CSS.[35]

An excellent example of procedural UI creation is Java Swing (figure 2.2) or JavaScript's Document Object Model manipulation (figure 2.3). Instead of describing how the view should look, a code is written to tell the interpreter

## HTML

```
<div id="main">
  <div id="toolbar">
    <button>
      </img>
      Cut
    </button>
  </div>
  <textarea id="editor"></textarea>
</div>
```

## Java Swing

```
JPanel main = new JPanel();

JPanel toolbar = new JPanel();
JButton button = new JButton();
button.setIcon(...);
button.setLabel("Cut");
toolbar.add(button);
main.add(toolbar);

JTextArea editor = new JTextArea();
main.add(editor);
```

Figure 2.2: Declarative HTML vs. procedural Java[35]

## Angular + HTML

```
<form id="main">
  <button mat-icon-button aria-label="Cut">
    <mat-icon>content_cut</mat-icon>
  </button>
  <mat-form-field id="editor">
    <textarea matInput [(ngModel)]="content">
    </textarea>
  </mat-form-field>
</form>
```

## Document Object Model (DOM) in Javascript

```
var main = document.createElement("div");
main.setAttribute("id", "window");

var toolbar = document.createElement("div");
toolbar.setAttribute("id", "toolbar");

var button = document.createElement("button");
var img = document.createElement("img");
img.setAttribute("src", "cut.png");
button.appendChild(img);

var label = document.createTextNode("Cut");
button.appendChild(label);
toolbar.appendChild(button);
window.appendChild(toolbar);

var editor = document.createElement("textarea");
editor.setAttribute("id", "editor");
window.appendChild(editor);
```

Figure 2.3: Declarative Angular HTML vs. procedural DOM[35]

precisely what to do (create an element, insert an element into another one, append an element to the end of the list). The result of both of these approaches can be the same. However, the declarative approach is more abstract, and it is easier to change and develop the UI since it is easier to understand, and parts of the code can be made into templates and reused in other projects.[35]

While HTML, Java Swing, and JavaScript DOM manipulation are great and clear examples of the main differences between procedural and declarative UI, most of these are surpassed and are not used in modern web development anymore. In the modern web, it is common to combine declarative UI with the MVVM pattern because the MVVM provides two-way data binding, allowing easy dynamic data display and state changes in the declared view. Many frameworks, such as React, Angular, and Blazor, use modified HTML to define a view and support two-way data binding between the view and the underlying view model. These components then use dependency injection to get required data, closely resembling the MVVM pattern.

## 2.4 Domain-specific languages for declarative user interface

In the book *Domain-Specific Languages*[36], Martin Fowler defines DSL as follows:

*DSL is a programming language of limited expressiveness focused on a particular domain.*

The definition consists of four key elements:

**Computer programming language** Humans use a DSL to instruct a computer to do something. Its structure is designed to make it easy for humans to understand, but it should still be executable by a computer.

**Language nature** A DSL should have a sense of fluency. The expressiveness comes not only from individual expressions but also from the composition of these elements.

**Limited expressiveness** A general-purpose programming language provides many expressions, which makes it harder to learn and use. A DSL supports a bare minimum of features needed to support its domain. A DSL should be used for one particular aspect of a system, not the entirety of it.

**Domain focus** A limited language is only applicable if it focuses on a small domain. The focus is what makes it worthwhile.[36]

The DSLs can be further divided into internal and external DSLs. Internal DSL is based on using a subset of a general-purpose language in a way

that feels like a custom language. A typical example of this is Ruby and its frameworks (for example, Rails), which use Ruby and its duck-typing to solve a specific problem. External DSL is a language separate from the main language of the application. It has a custom syntax or uses a syntax of another language. The most common example of this is XML, which is used as a language for many other DSLs. The most important feature of an external DSL is its clarity for the reader. It should be designed so that anyone with programming knowledge can understand it just by looking at it. On the other hand, it should not try to mimic natural language, as it adds a lot of syntactic sugar, which makes the semantics harder to understand, and after all, the DSL is intended for programmers.[36]

### 2.4.1 XML based domain-specific languages

Extensible Markup Language is a markup language. It is a syntactic structure without semantics, and many external DSLs use it as a carrier syntax. Many project configuration files could be looked at as DSLs (not to confuse with property lists, which serve only for storing key-value pairs with a structure to them).[36]

XMLs main disadvantage is its rather noisy syntax. There are many angle brackets, quotes, slashes, and its nesting elements require opening and closing tags, resulting in a poor text to actual content ratio. However, XML has its advantages too. It strongly resembles HTML; most programmers are familiar with its format and can read it, and there are many available parsers and other tools for working with the format.[36]

XML is a good carrier syntax candidate for a DSL that will define DApps UI, mainly because of its HTML resemblance and the fact that Das Contract mainly consists of forms. Additionally, XML comes with technologies that allow checking the XML without executing it by comparing it to a schema; this might come in handy when validating the forms.

### 2.4.2 Domain-specific languages and declarative user interface

Extensible Application Markup Language (XAML) is an XML implementation for specifying user interfaces. It is used to organize passive objects into a structure. XAML is an excellent example of a DSL, which uses declarative programming to layout UI without any reference to control flow, separating the screen layout from code. MVVM pattern can then be used to connect the XAML-based view with the underlying view model, allowing it to fill the view with data from the model using data binding.[33][36]

XAML has been used extensively by Microsoft, and it has been adapted to several technologies within the .NET Framework. It has also been used for defining the layout of UIs within the Windows Presentation Foundation,

Silverlight, the Windows Runtime, the Universal Windows Platform, and Xamarin Forms. It is well suited to be used with the MVVM pattern for defining UIs declaratively.[37]

## 2.5 Interaction Flow Modeling Language

The Interaction Flow Modeling Language (IFML) is a modeling language for defining platform-independent descriptions of graphical user interfaces, supporting desktop, laptop, phone, and even tablet and PDA systems. The description of the application UI focuses on the structure and behavior as perceived by the end-user and is limited to aspects that directly influence the user's experience.[38] In other words, the IFML describes the view of the MV\* pattern family with a single Interaction Flow Diagram in the following perspectives:

**The view structure specification** consists of the definition of view containers, their nesting, visibility, and reachability.

**The view content specification** consists of the definition of view components (content and data contained within view containers).

**The events specification** consists of the definition of events that affect the UI's state (events produced by user interaction, application, or an external system).

**The event transition specification** consists of defining an event's effects on the UI (dynamic change of the displayed content and actions being triggered).

**The parameter binding specification** consists of defining the input-output dependencies between view components (data propagation and flow within the view).[38]

Using a platform-independent model brings several benefits. It allows explicit representation of the front-end's different perspectives (content, interface organization, interaction, navigation, connection with business logic). It raises the abstraction level of the front-end specification, isolating it from implementation specifics and allowing better coordination of work in the development process. It enables communication of the UI and its interactions to stakeholders through the Interaction Flow Diagram or its automatic generation into UI, allowing early validation of business requirements.[39]

## 2.6 Chapter summary

This chapter has shown a possible mapping of the three-tier architecture to semi-decentralized applications and how we can view the presentation layer as if it was an enterprise application. We have concluded that the MVVM pattern is the best candidate from the MV\* pattern family to be used in decentralized applications. We have also shown how we can use external domain-specific languages to declare user interfaces, either through XML-based formats or other platform-independent models such as IFML. We will use most of these findings in the next chapter to propose a way to define user interfaces of decentralized applications in a platform-independent manner.



---

# Towards a new domain-specific language for decentralized application's user interface

This chapter shows and analyzes two possible approaches to defining decentralized application's user interface. Afterward, requirements for the domain-specific language are described, and based on those requirements, a new forms model is proposed, and its functionalities are explained. Based on the forms model, a new domain-specific language is introduced, meeting the specified requirements. The end of the chapter describes how the forms model can be converted into the previously examined notations.

## 3.1 First look at the user interface

At the end of chapter one, we have introduced the Das Contract and its mortgage case study. In the following chapters, we will use the mortgage case to demonstrate different approaches to creating the Das Contract UI. Figures 3.1, 3.2, 3.3, and 3.4 show wireframes of the Das Contract wallet. The wireframes should help us identify which elements need to be supported by the DSL and give us a general idea of how the application should function and look.

Figure 3.1 shows the dashboard of the application. The dashboard shows activities that the user needs to do based on their due date. Everything we can see in figure 3.1 should not be described by the DSL but created based on the current contracts the user is subscribed to. The left menu shows all contracts the user currently deals with and allows searching within them, creating new ones, and joining existing ones. The dashboard shows activities from those contracts that have the earliest due date.

Figures 3.2, 3.3, and 3.4 show activities the borrower role can perform in

### 3. TOWARDS A NEW DOMAIN-SPECIFIC LANGUAGE FOR DECENTRALIZED APPLICATION'S USER INTERFACE

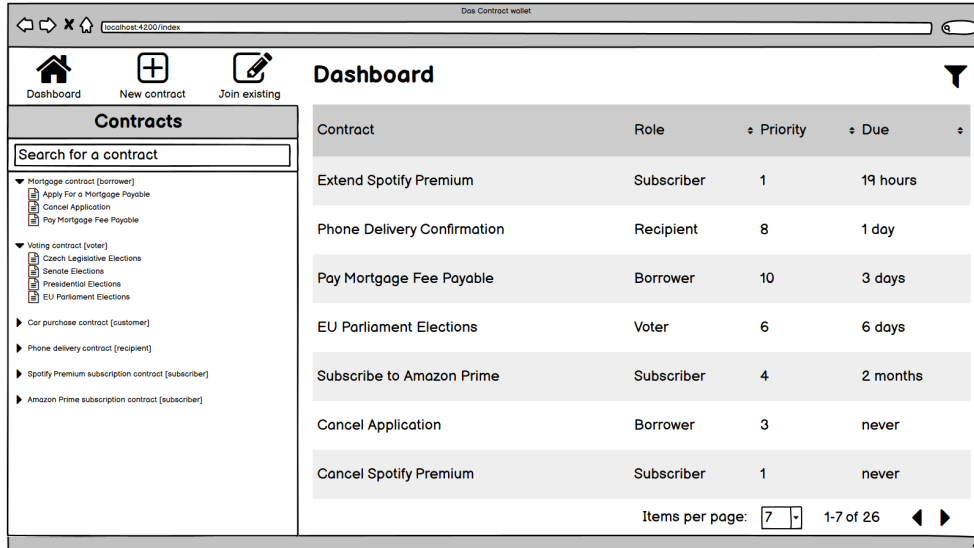


Figure 3.1: Wireframe of the Das Contract wallet dashboard

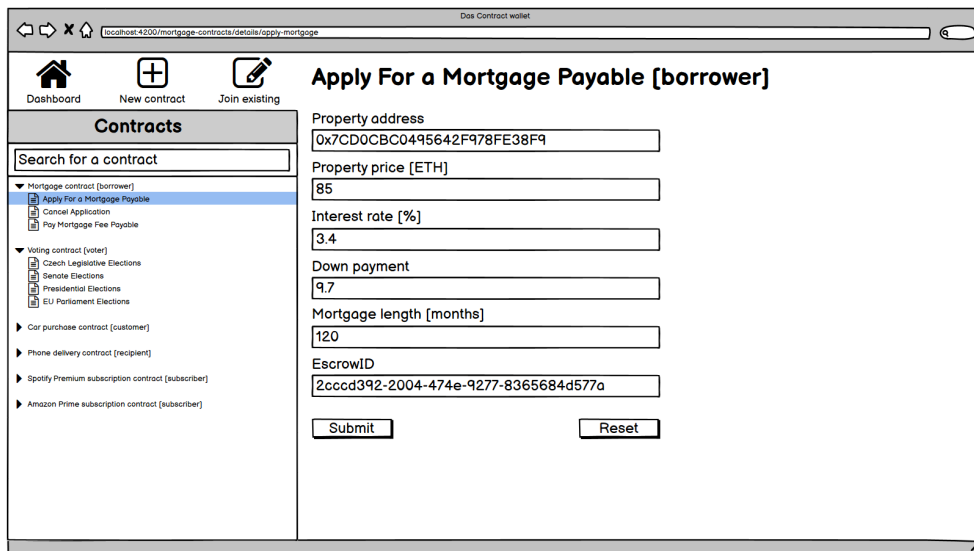


Figure 3.2: Wireframe of the Das Contract mortgage contract - apply for a mortgage

### 3.1. First look at the user interface

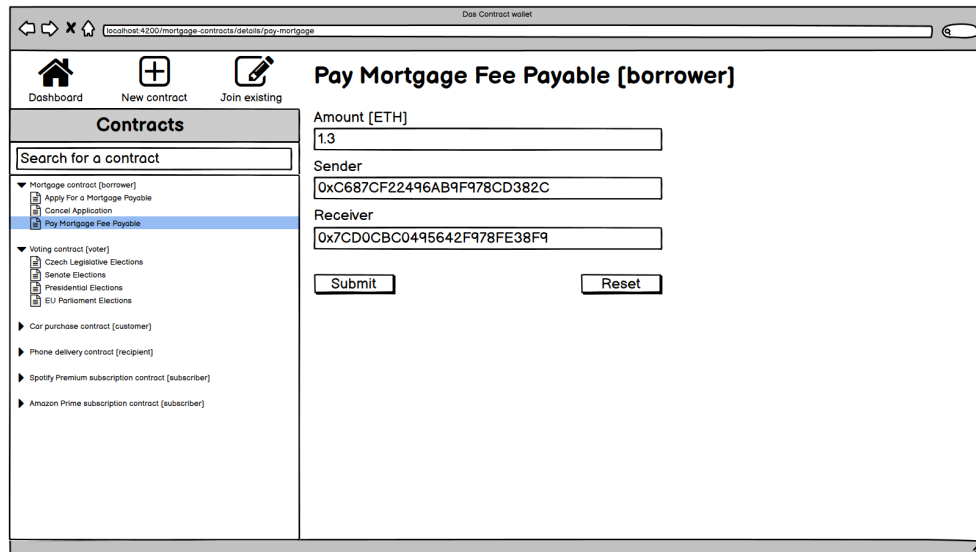


Figure 3.3: Wireframe of the Das Contract mortgage contract - pay for a mortgage

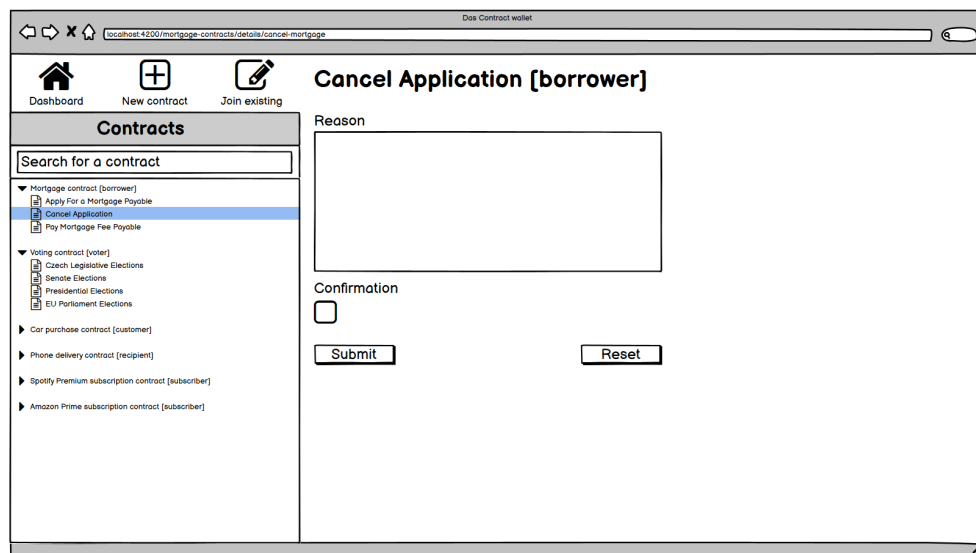


Figure 3.4: Wireframe of the Das Contract mortgage contract - cancel a mortgage

the Das Contract mortgage contract. The borrower can apply for a mortgage, pay for a mortgage and cancel a mortgage application. Each of these activities has its form the user needs to fill. The proposed DSL should allow creating these forms in a platform-independent way. According to the Das Contract paper[25], the forms only support primitive data types such as int, bool, double, string, and more complex ones like enums, arrays, time, and blockchain addresses.

### 3.2 Creating Das Contract forms with Interaction Flow Modeling Language

In chapter two, we had a look at IFML and its UI modeling capabilities. IFML is an OMG standard for modeling UI into platform-independent models. It also supports UI generation from the platform-independent model, and so, might seem like a good candidate for our DSL if we cut out things we need out of it.

The IFML packs the entire model into a single diagram called the Interaction Flow Diagram, unlike UML, which is split across many purpose-specific diagrams.[39] While it is nice to have everything in one place, it also brings increased complexity, and many elements and concepts are being mixed. Figure 3.5 shows a subset of elements from the IFML notation that could be used for a DSL to represent DApps UI. Figure 3.6 shows a simplified representation of the Das Contract wallet dashboard in the IFML notation (seen before as wireframe in figure 3.1). Figure 3.7 shows how the IFML model looks when converted into code by IFMLEdit (note that IFML does not support additional graphical information, and therefore, the wireframe looks quite different from the generated code).

Using a subset of IFML as DSL for modeling UI would allow us to create a platform-independent model, which can then be used for generating code of the application. Several tools that can generate code from IFML already exist[40]; however, they usually have limited support for the notation, hinting that it might be challenging to implement correctly and according to the standard. Additionally, the IFML standard is created to do what UML does, except for the UI. It purposely disallows presentation aspects in its diagram because it does not adhere to the abstraction level the model aims to provide. The main goal is modeling UI, thus documenting it for other developers in a group project and allowing an easier understanding of the UI and interaction flow. The code generation and model interpretation, which is the core requirement for our use-case, is more of a side effect, similar to using UML diagrams to generate code. The primary purpose of generating code from UML diagrams is to quickly provide a skeleton of the project we can build upon, whereas we need a DSL to write simple UI with forms.[39][38]

### 3.2. Creating Das Contract forms with Interaction Flow Modeling Language

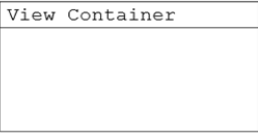
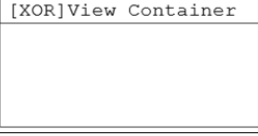
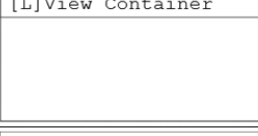
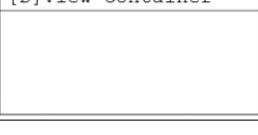

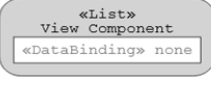

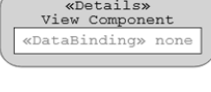


Element type	Meaning	IFML notation	Implementation example
View container	An element of the interface that comprises elements displaying content and other view containers		HTML element
XOR view container	A view container comprising child view containers that are displayed alternatively		Nested <div> element made visible on-demand
Landmark view container	A view container that is reachable from any other element of the user interface without having explicit interaction flow		HTML anchor
Default view container	A view container that will be presented by default to the user, when its enclosing container is accessed		Nested <div> element that is visible by default
View component	An element of the interface that displays content or accepts input		Input form
List	View component used to display a list of data-binding instances		Table with rows that have the same element kind
Form	View component used to display a form that is composed of fields		HTML form
Details	View component used to display details of a specific data-binding instance		<div> element with content based on the data-binding
Action	A piece of business logic triggered by an event		<script> with business logic
Event, submit event, select event	An occurrence that affects the state of the application, parameter passing between elements or selection of a single item of the user interface		User click, form submission, selection of a row in a table

Figure 3.5: Subset of IFML elements that could be used for DApps UI[38]

### 3. TOWARDS A NEW DOMAIN-SPECIFIC LANGUAGE FOR DECENTRALIZED APPLICATION'S USER INTERFACE

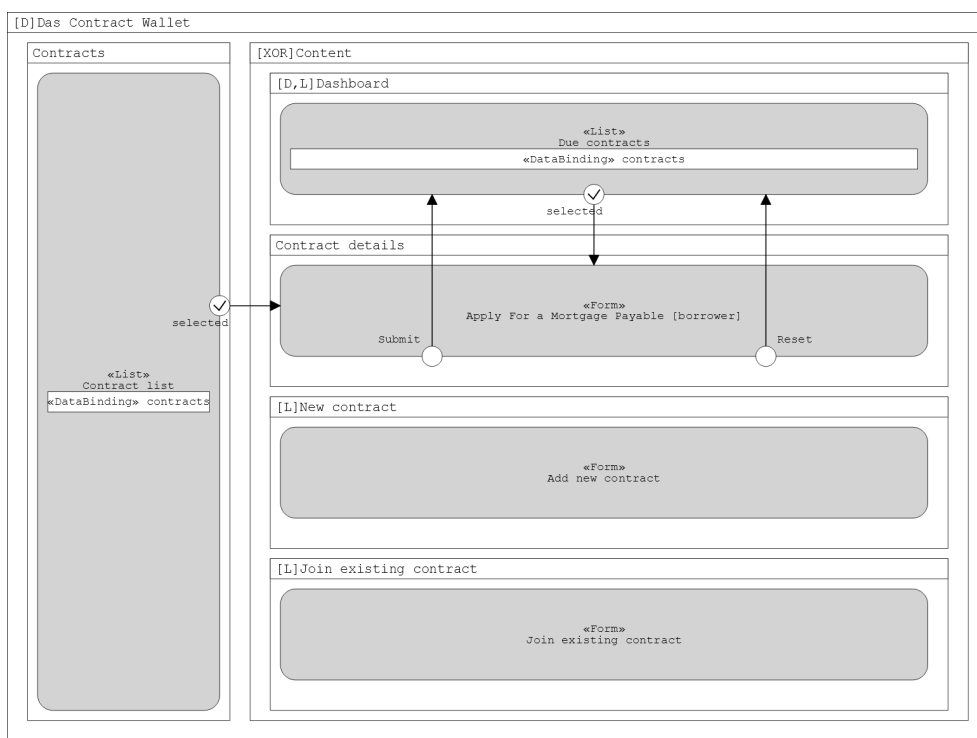


Figure 3.6: IFML diagram of the Das Contract wallet dashboard designed in IFMLEdit[40]

#### 3.2.1 Conclusion

IFML is a powerful UI modeling language. It allows modeling the UI of an entire application along with its data bindings and navigation flow. IFML needs to be very complex to fulfill its role, making it a bad fit for our use-case of modeling smart contract forms. Additionally, the code generation is something extra the IFML offers, rather than being the primary concern. We should look for simple solutions that are easily customizable for our needs and are a better fit for generating UI.

### 3.3 Creating Das Contract forms with XML-based domain-specific language

In chapter 2, we also had a look at domain-specific languages that use XML as carrier syntax. XML's main disadvantage is its noisy syntax. Other than that, it has excellent language support (many parsers exist in every programming language, it allows validation through XML schemes, and it has XPath and XQuery, which are potent languages for navigating and querying XML

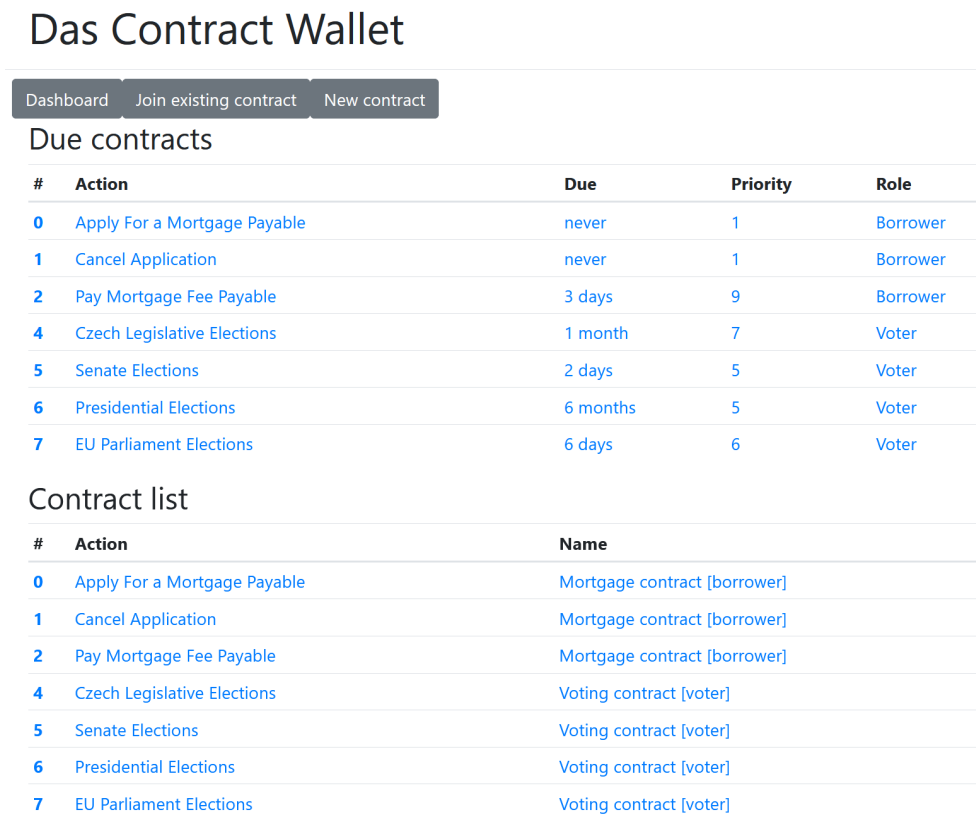


Figure 3.7: UI of the Das Contract wallet dashboard generated from IFML by IFMLEdit[40]

documents). Additionally, XML syntax has been used to specify user interfaces before in the .NET framework, it is very similar to HTML, and many web developers are familiar with it.

To see if XML is a suitable syntax, we can try modeling the Das Contract apply for a mortgage form shown in Figure 3.2. The left part of figure 3.8 shows how this form could look in XML syntax. Most software developers should see that the form has a name and six fields the user can fill without any prior knowledge of our made-up syntax. Therefore, the syntax would be easy to learn and understand at first sight. However, from the 24 lines in the XML document, only 7 carry unique information, such as the field names, making the noise prevalent.

If we add some additional attributes, such as data type, or field description, we will get more content over the noise, as seen in the middle part of figure 3.8, but we can reduce the noise even further. Since the language is specific to our domain, we can start bending the XML syntax to our particular use. First,

### 3. TOWARDS A NEW DOMAIN-SPECIFIC LANGUAGE FOR DECENTRALIZED APPLICATION'S USER INTERFACE

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <Form>
3   <Name>Mortgage form</Name>
4   <Fields>
5     <Field>
6       <Name>Property address</Name>
7     </Field>
8     <Field>
9       <Name>Property price (ETH)</Name>
10    </Field>
11    <Field>
12      <Name>Interest rate (%)</Name>
13    </Field>
14    <Field>
15      <Name>Down payment</Name>
16    </Field>
17    <Field>
18      <Name>Mortgage length (months)</Name>
19    </Field>
20    <Field>
21      <Name>Escrow ID</Name>
22    </Field>
23  </Fields>
24 </Form>

```

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <Form>
3   <Name>Mortgage form</Name>
4   <Fields>
5     <Field>
6       <DataType>address</DataType>
7       <Name>Property address</Name>
8     </Field>
9     <Field>
10      <DataType>currency</DataType>
11      <Name>Property price (ETH)</Name>
12    </Field>
13    <Field>
14      <DataType>decimal</DataType>
15      <Name>Interest rate (%)</Name>
16    </Field>
17    <Field>
18      <DataType>decimal</DataType>
19      <Name>Down payment</Name>
20    </Field>
21    <Field>
22      <DataType>integer</DataType>
23      <Name>Mortgage length (months)</Name>
24      <Description>How many month you'll pay.</Description>
25    </Field>
26    <Field>
27      <DataType>string</DataType>
28      <Name>Escrow ID</Name>
29      <Description>Identifier for the mortgage escrow.</Description>
30    </Field>
31  </Fields>
32 </Form>

```

Figure 3.8: Example of syntactic noise in the apply for a mortgage form

```

1 <Form>
2   <Name>Mortgage form</Name>
3   <Field>
4     <DataType>address</DataType>
5     <Name>Property address</Name>
6   </Field>
7   <Field>
8     <DataType>currency</DataType>
9     <Name>Property price (ETH)</Name>
10  </Field>
11  <Field>
12    <DataType>decimal</DataType>
13    <Name>Interest rate (%)</Name>
14  </Field>
15  <Field>
16    <DataType>decimal</DataType>
17    <Name>Down payment</Name>
18  </Field>
19  <Field>
20    <DataType>integer</DataType>
21    <Name>Mortgage length (months)</Name>
22    <Description>How many month you'll pay.</Description>
23  </Field>
24  <Field>
25    <DataType>string</DataType>
26    <Name>Escrow ID</Name>
27    <Description>Identifier for the mortgage escrow.</Description>
28  </Field>
29 </Form>

```

```

1 <Form>
2   <Name>Mortgage form</Name>
3   <AddressField>
4     <Name>Property address</Name>
5   </AddressField>
6   <CurrencyField>
7     <Name>Property price (ETH)</Name>
8   </CurrencyField>
9   <DecimalField>
10    <Name>Interest rate (%)</Name>
11  </DecimalField>
12  <DecimalField>
13    <Name>Down payment</Name>
14  </DecimalField>
15  <IntegerField>
16    <Name>Mortgage length (months)</Name>
17    <Description>How many month you'll pay.</Description>
18  </IntegerField>
19  <StringField>
20    <Name>Escrow ID</Name>
21    <Description>Identifier for the mortgage escrow.</Description>
22  </StringField>
23 </Form>

```

Figure 3.9: Removing syntactic noise to increase information value

```

1 <Form Name="Mortgage form">
2   <AddressField Name="Property address" />
3   <CurrencyField Name="Property price (ETH)" />
4   <DecimalField Name="Interest rate (%)" />
5   <DecimalField Name="Down payment" />
6   <IntegerField Name="Mortgage length (months)" Description="How many month you'll pay." />
7   <StringField Name="Escrow ID" Description="Identifier for the mortgage escrow." />
8 </Form>

```

Figure 3.10: Apply for a mortgage form with minimal syntax noise



we can remove the unnecessary encoding meta-information and the `Fields` tag, which is meant to surround array elements. Doing this will get us to the left part of figure 3.9. Now, the most significant noise consists of each element being a pair of opening and closing tags. Additionally, the `Field` tag, which only works as a surrounding tag, does not carry any information.

We can replace the `Field` tag by making it into a field specialization, representing input data type. Such change removes the need for a `DataType` element, and it also makes the syntax closer to how HTML handles element types. The result can be seen in the right part of figure 3.9. Finally, to get rid of the closing tags, we can define the properties as attributes instead of a standalone element, which will also allow us to use the short notation for XML tags. The final result can be seen in figure 3.10. We have successfully removed almost all the XML syntax noise while keeping the XML format standard enough to resemble HTML and be supported by common XML parsers and query languages.

#### 3.3.1 Conclusion

Using XML-based DSL to specify user interfaces has many advantages, such as similarity to HTML, easy understanding, and existing software support. XML's main issue is significant syntax noise overhead, which we were able to deal with by modifying the syntax without removing its advantages. Therefore, we will use the modified XML as a carrier syntax for our domain-specific language.

## 3.4 Requirements for Das Contract domain-specific language

In the previous chapter, we have pointed out that domain-specific languages are helpful only when they have a limited scope. Therefore, we need to define the required functionalities the DSL should support. The following subsections describe which functionalities the DSL should support, and together, they form the DSL's domain.

#### 3.4.1 Creating forms

As noted in chapter 1, the Das Contract consists of the process model, the data model, and the forms model. The core functionality of the DSL is to allow creating the user interface for the forms model. The form is bound to a user task, and it should allow the user to accomplish his task in the smart contract through the form. The form can be divided into multiple sections called field groups for clarity, and each field group can have various fields.

### **3.4.2 Collecting user input**

Most user tasks require user input. The form should collect this input, format it so that the smart contract understands it, and send it to the smart contract. The form collects user input through its fields. As mentioned at the start of this chapter, there is a limited amount of supported data types: Address, String, Date and Time, Integer, Decimal, Bool, Array, and Enum. To collect all user inputs correctly, the DSL should gather all of the above data types in a user-friendly way and send them to the smart contract.

### **3.4.3 Display Das Contract details**

Since we expect the user to interact with the smart contract, the DSL should support providing feedback to the user. Therefore, the DSL should support pulling information from the blockchain and displaying it to the user to help him decide on his actions.

### **3.4.4 Validate user task roles and control flow**

The Das Contract is defined by a limited BPMN diagram, which controls the flow of user tasks and activities in general. Additionally, each user task has a role that can complete this task. While the generated smart contract checks this for us and will reject a transaction if it is not valid, we want to make the user experience better. Therefore, the DSL should only display forms the current user can complete at the given contract state to guide the user through the contract.

### **3.4.5 Basic support for smart contracts outside of the Das Contract methodology**

To complete the user tasks, the user might sometimes need to interact with other smart contracts that were not made with the Das Contract methodology (for example, interacting with a property token for a mortgage contract). While this thesis's primary goal is to define the UI of Das Contract, the DSL can go a bit further and allow defining the UI of any smart contract available on the blockchain. While the interaction will be limited to forms only, it is often enough to accomplish essential smart contract interactions.

### **3.4.6 Easy to extend in the future**

The Das Contract methodology has been in development for multiple years, and it has changed a lot from its initial state. Since there are plans to develop this methodology further, the DSL should be easily extendable if new requirements arise (for example, if new data types were needed).

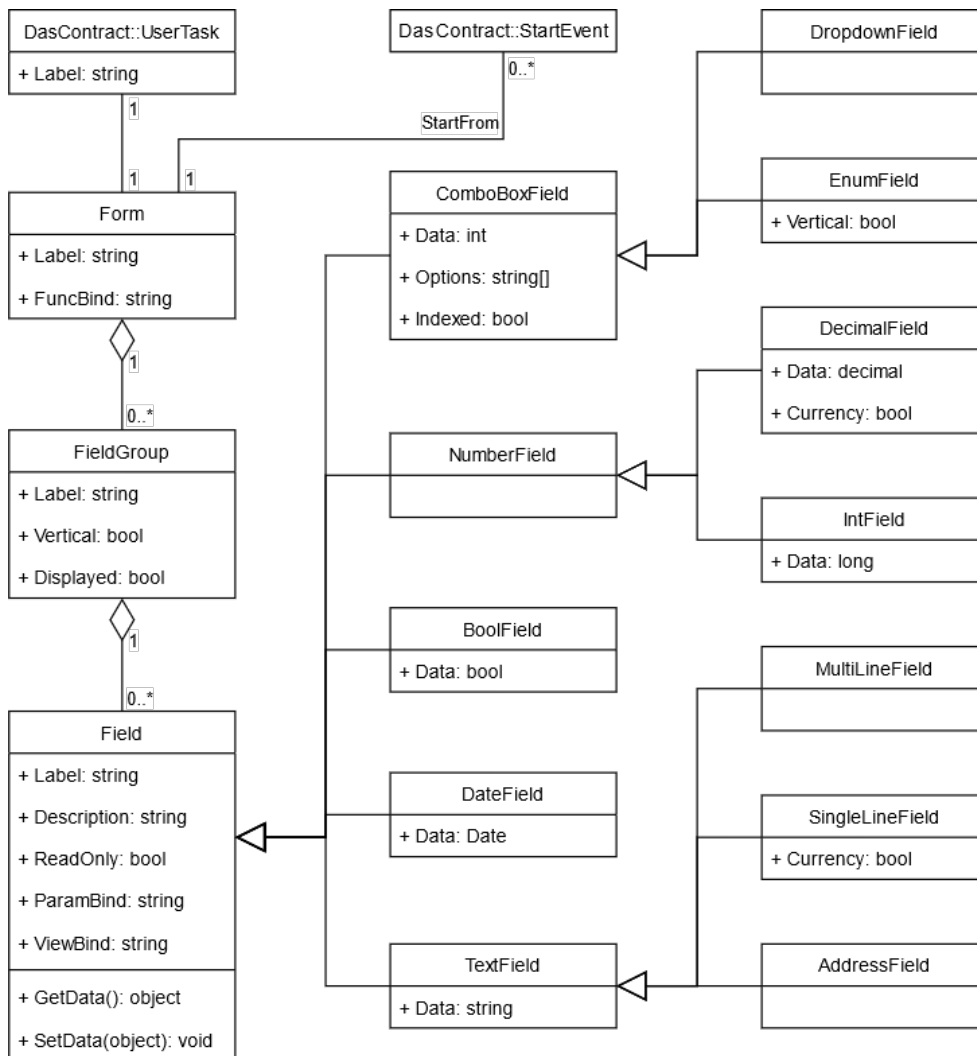


Figure 3.11: New Das Contract forms model

### 3.5 Redefining Das Contract forms model

Forms model specifies a user form required to fill by the user in a process user task. The forms provide a way to interact with smart contracts.[26] Figure 3.11 shows a new forms model designed to be part of the Das Contract metamodel shown in figure 1.8. It is designed to fulfill the requirements specified in the previous section and allow seamless interaction with a blockchain focusing on user experience.

### 3. TOWARDS A NEW DOMAIN-SPECIFIC LANGUAGE FOR DECENTRALIZED APPLICATION'S USER INTERFACE

---

In the new forms model, the Form<sup>1</sup> consists of FieldGroups, which consist of Fields. The Form holds Label as its attribute, which serves as a display name, and a FuncBind, which allows interaction with smart contracts outside of the Das Contract methodology. The FieldGroup is meant to separate groups of fields that serve one functionality (for example, personal details). Separating fields by functionality allows the user to accomplish his goals and navigate within the form faster. The FieldGroup can have a Label, which should tell the user the purpose of the fields. Its fields can be rendered vertically (separated by a newline) or horizontally (next to each other) based on the Vertical attribute. The group can also be hidden and shown on-demand based on the Displayed attribute to display new details to fill in gradually.

The Field represents an abstract class that each input element must inherit. This approach allows anyone to easily extend the forms UI by simply implementing the Field methods and defining how it should be rendered into HTML by the interpret. A Field must allow getting and setting the input data through a generic object. If the provided object cannot be converted to the data type the field works with, an exception should be thrown. The Field has a Label attribute to tell the user what should be filled in. It can have a Description attribute, which is used as a tooltip if a user needs help filling in the field, and it can have a ReadOnly attribute, which specifies if the user can modify the field data.

The remaining two attributes in the Field class are bindings. The ViewBind allows retrieving information from the smart contract deployed on the blockchain and will be explained later. The ParamBind is used to bind input data into the smart contract transaction call parameters. If we interact with a smart contract transaction that takes parameters named *first* and *second*, the Form has to contain two Fields, one having ParamBind set to *first* and the other to *second*. The ParamBind can have a special construct: *const:payValue*. If a Field contains this value, it means that the Field contains an amount of cryptocurrency to be sent to the smart contract. This construct allows executing smart contract transactions that require payment to be made to the contract address.

Several specializations of the Field class exist. These specializations are meant to cover the basic data types declared in the previous section. If a field contains a Currency attribute, it means that the data should be formatted according to the blockchain's base currency (for example, for Ethereum, the data coming from the blockchain are in Wei, but they should be converted into Ether for user convenience). The ComboBoxField has an Options attribute, which can be used to declare fixed values the user can select from (for example, male or female). Also, the Indexed attribute is true, the field returns the index of selected data instead of returning the selected value, which is used to pass

---

<sup>1</sup>In this text, Form refers to the Form class from the new forms model, whereas form refers to an element that allows collecting data from the user.

selected value from an enum.

### 3.6 Domain-specific language requirements evaluation

Section 3.4 specifies essential requirements our DSL needs to support. This section explains how the new forms model helps us meet the specified requirements. It also provides insight into how the features are expected to behave when implemented.

#### 3.6.1 Creating forms

The forms model allows the designer to create a broad scope of forms. Each form can be separated into multiple sections through FieldGroups. Each section can have fields laid out vertically or horizontally. Each element in the form can have its label, which helps the user navigate the form faster. Each instance of the forms model is bound to one smart contract (usually tied to a Das Contract through the UserTask), meaning one form cannot communicate with multiple deployed smart contracts.

#### 3.6.2 Collecting user input

Each Field has to implement the GetData and SetData methods, allowing parsing user input into smart contract format. Many specializations of the Field class exist to support the required data types. Besides the apparent specializations, the MultiLineField can display big blocks of text such as descriptions or string arrays. The DropDownField can be used to select one from many, and the EnumField can be used to select one from a few. The Field specializations provide a user-friendly interface because each specialization can be rendered as a different HTML element. This way, the user does not have to fill in all data into a simple input text field and can instead be presented with various input options.

#### 3.6.3 Display Das Contract details

Each Field can have a ViewBind attribute, which allows the designer to pull data from the smart contract on the blockchain and present them to the user. Since each Form is bound to one smart contract, the Field can only pull data from this contract. The ViewBind follows a format similar to an object-oriented language but is severely limited.

The ViewBind format is as follows: *name(param1, param2, ...).attribute [index]*. The only required symbol is the name, which specifies which function or public attribute to fetch from the smart contract. If the name refers to a smart contract call, parameters can be provided to pass into the function

### 3. TOWARDS A NEW DOMAIN-SPECIFIC LANGUAGE FOR DECENTRALIZED APPLICATION'S USER INTERFACE

---

```
Mortgage public mortgage;           // Values
bool public isCompleted;             // false
struct Mortgage {
    uint256 propertyId;               // 0
    uint256 propertyPrice;           // 11000
    uint256 rate;                     // 5
    uint256 mortgageDurationMonths;  // 60
    uint256[] deposits;              // [20, 30, 40]
    Payment downPayment;             // Defined below
}
struct Payment {
    uint256 amount;                   // 35
    bool onSchedule;                  // true
}
function getDownPayment() public view returns (Payment) {
    return mortgage.downPayment;
}
function getDeposit(uint256 index, bool includePremium)
    public view returns (uint256) {
    if (includePremium) {
        return deposits[index] * (1 + mortgage.rate / 100);
    }
    return deposits[index];
}

ViewBind calls:                       // Values retrieved
isCompleted                            // false
isCompleted()                          // false
mortgage.propertyPrice                  // 11000
mortgage().propertyPrice                // 11000
mortgage.deposits[0]                    // 20
mortgage.deposits                       // [20, 30, 40]
getDownPayment().amount                 // 35
getDownPayment.amount                   // 35
getDeposit(0, true)                     // 21
getDeposit(0, false)                    // 20
```

Figure 3.12: ViewBind format examples

call. If there are no parameters or the name refers to a public attribute, the *(param1, param2, ...)* can be completely left out. If the data received from the call are structured, we can use *.attribute* to access a specific attribute within those data. The attributes cannot be chained (for example, *mortgage.payment.amount* cannot be used to access a structure within a structure), and if such access is needed, the smart contract should provide a getter for it. If the received data or attribute are enumerable, *[index]* can point to a specific position within those data. If an index is not provided, the entire array is returned, and appropriate Field specialization should be used (DropDownField, EnumField, or MultiLineField). Figure 3.12 shows an example of Solidity smart contract data and how ViewBind can be used to access those data.

The ViewBind format can also be *const:name*. If that is the case, the name either refers to a known constant or a filler value. In a ViewBind, there are currently two supported constants: *myAccountAddress*, which refers to the currently active account address, and *myContractAddress*, which refers to the deployed smart contract address. If the name refers to anything else, it will be used as a value set into the Field when the form is loaded. This is most useful with the *ReadOnly* attribute, as the designer can define constant values the user cannot change (for example, the designer can set an interest rate field to *const:5*, which will fix the applied for mortgage rate to 5%).

#### 3.6.4 Validate user task roles and control flow

Every Form is associated with a single UserTask from the Das Contract. Since UserTask is a specialization of the ProcessElement, it is easy to see if the UserTask can be currently completed using the SequenceFlow. The UserTask has CandidateRoles, such as borrower, lender, insurer, and property owner, who possess the right to execute this UserTask. When it comes to implementation, both role and current flow state need to be retrieved from the blockchain. Multiple participants can operate the smart contract, which can move the sequence forward, and also new addresses can enter the contract and get a role assigned. In the generated Das Contract, the *addressMapping* property provides a role to address mapping, and the *ActiveStates* property tells us whether the ProcessElement is currently executable.

Basic support for smart contracts outside of the Das Contract methodology Fields can pull data from any smart contract using the ViewBind attribute, which is not exclusive to Das Contract. Suppose the application wants to send data to the smart contract (execute a transaction). In that case, it is not straightforward since the Form is supposed to be bound to a Das Contract through UserTask, which provides the application with function names for transaction execution. If a designer wants to execute a transaction on a generic smart contract, he can provide the Form with a FuncBind, which tells what

### 3. TOWARDS A NEW DOMAIN-SPECIFIC LANGUAGE FOR DECENTRALIZED APPLICATION'S USER INTERFACE



Figure 3.13: A mapping of the forms model into the XML syntax

function to execute within the smart contract. The Field's ParamBind should match the function arguments so that the data are passed correctly.

These three bindings together allow complete communication with generic smart contracts, as long as the supported Fields can represent the data. Further research on this is out of the scope of this thesis, but the main functionality is to communicate with smart contracts that work along with Das Contract (for example, working with tokens that can be used in Das Contract).

#### 3.6.5 Easy to extend in the future

The forms model was designed so that new Field specializations can be added if needed to be. Adding a new Field specialization is as easy as creating a class implementing the abstract getter and setter and telling the application renderer how to render such Field into HTML. Also, since communication with generic smart contracts is supported, it should always be possible to execute Das Contract even if a new version is released.



### 3.7. Forms model as a domain-specific language

---

```
<Form Label="Apply for a mortgage" FuncBind="ApplyForAMortgagePayable">
  <FieldGroup Label="Submit mortgage application">
    <IntField Label="Property ID" ParamBind="propertyId" Description="ID of an ERC721 HouseToken that you are interested in." />
    <SingleLineField Label="Property price (Ether)" Currency="true" ParamBind="propertyPrice" />
    <IntField Label="Mortgage rate (%)" ParamBind="rate" />
    <IntField Label="Mortgage duration (months)" ParamBind="mortgageDurationMonths" />
    <SingleLineField Label="Mortgage downpayment (Ether)" Currency="true" ParamBind="const:payValue" />
  </FieldGroup>
</Form>
```

Figure 3.14: A form of the apply for a mortgage activity in the mortgage contract

## 3.7 Forms model as a domain-specific language

The forms model is sufficient for creating decentralized application's user interface. However, there has to be a way to instantiate it at runtime for interpretation to the user. In section 3.3, we have defined an XML format that we can use as a carrier syntax for our domain-specific language. The advantage of this format is that it has minimal syntax noise, making it easy to work with. Also, the forms model is represented by a class diagram, which can be easily converted into classes in an object-oriented language. These classes can then be instantiated by deserializing the XML into them, the only difference being the slight XML format adjustments.

Figure 3.13 shows how the forms model is mapped into the proposed XML syntax. Figure 3.14 shows the apply for a mortgage form designed in the XML syntax. The whole mortgage contract form can be found on Github<sup>2</sup>.

## 3.8 Chapter summary

This chapter concluded that the Interaction Flow Modeling Language is not a good fit for defining decentralized application's user interface. Instead, we have proposed an XML-based domain-specific language that should be easy to understand without any prior knowledge. The domain-specific language is based on the new proposed forms model and uses XML as its carrier syntax. The syntax rules were adjusted for minimal overhead and maximal information value. The domain-specific language should fully support Das Contract and also partially generic smart contract interaction.

---

<sup>2</sup><https://github.com/ancinpet/thesis-wallet/tree/main/src/thesis-wallet/Contracts/Forms>



---

# Proof of concept

This chapter describes the implementation process of the forms editor and forms wallet applications. In the beginning, the used technologies and the project scope and use cases are described. Afterward, both applications' functional and non-functional requirements are defined along with their architecture and design. Near the end, implementation details, testing, and deployment are described. In the end, the whole project is showcased, including a complete walkthrough of the Das Contract mortgage contract.

## 4.1 Used technologies

### 4.1.1 Blazor WebAssembly

Blazor is a single-page application (SPA) web framework, which allows building interactive UI in C# instead of JavaScript. There are currently two versions available: server-side and WebAssembly. The server-side version runs C# code on the server and communicates with the client through SignalR. The WebAssembly version is downloaded to the client when the web is first loaded, and after that, C# code is running directly in the browser. The server-side version requires a stable connection to the server but offers compatibility on older browsers since the C# code is running on the server and not in the browser. We will use the WebAssembly version for this project because once it is downloaded, the application can run without any connection to the server, meaning we do not rely on a central server in a decentralized application. Using Blazor allows us to use most of the .NET framework libraries and functionalities, such as Nethereum or various XML serializers.[41][42]

### 4.1.2 Nethereum

Nethereum is a .NET integration library for Ethereum. It simplifies interaction with the blockchain and smart contracts deployed on it.[43] It also offers

a Blazor interoperability template with MetaMask integration[44], which will be used in the forms wallet to communicate with the Ethereum network.

### 4.1.3 Monaco Editor

The Monaco Editor is Microsoft's code editor that powers VS Code. It supports the most significant web browsers, and it is licensed under the MIT license. It offers the same functionalities as VS Code in the web browser, giving us access to a full-featured code editor without implementing anything.[45] In this project, we will use BlazorMonaco[45], which is a Blazor component of the Monaco Editor. BlazorMonaco will allow us to use the Monaco Editor without worrying about JavaScript interoperability.

### 4.1.4 Material Design

Material is a design system created by Google to help teams build high-quality digital experiences for Android, iOS, Flutter, and the web. Material Design is inspired by the physical world and its textures, including how they reflect light and cast shadows. Material surfaces reimagine the mediums of paper and ink.[46]

Material Design will be used to create good looking and responsive UI. We will use MatBlazor, which provides Material Design components for Blazor.[47] It will allow us to use uniform input elements, menus, and buttons the user is used to from the modern web. Having the same look and feel as Google and other modern websites should make the user experience much better, and filling in the data should be more intuitive.

## 4.2 Project scope

In the previous chapter, we have defined the new forms model and how it is expected to behave. Figure 3.11 shows the forms model, and figure 3.13 shows how it is represented in an XML syntax. To make the model usable, we need to create two applications. The first application's purpose is to design the forms model in a user-friendly way. This application will be referred to as the forms editor. The second application will be referred to as forms wallet. Its primary purpose will be to interpret provided forms and allow the user to interact with Das Contract contracts and generic smart contracts deployed on the Ethereum blockchain. Both of these applications will use the forms model and its renderer as a shared functionality. The forms editor will use it to provide a visual preview of the designed form, and the forms wallet will use it as a user interface.

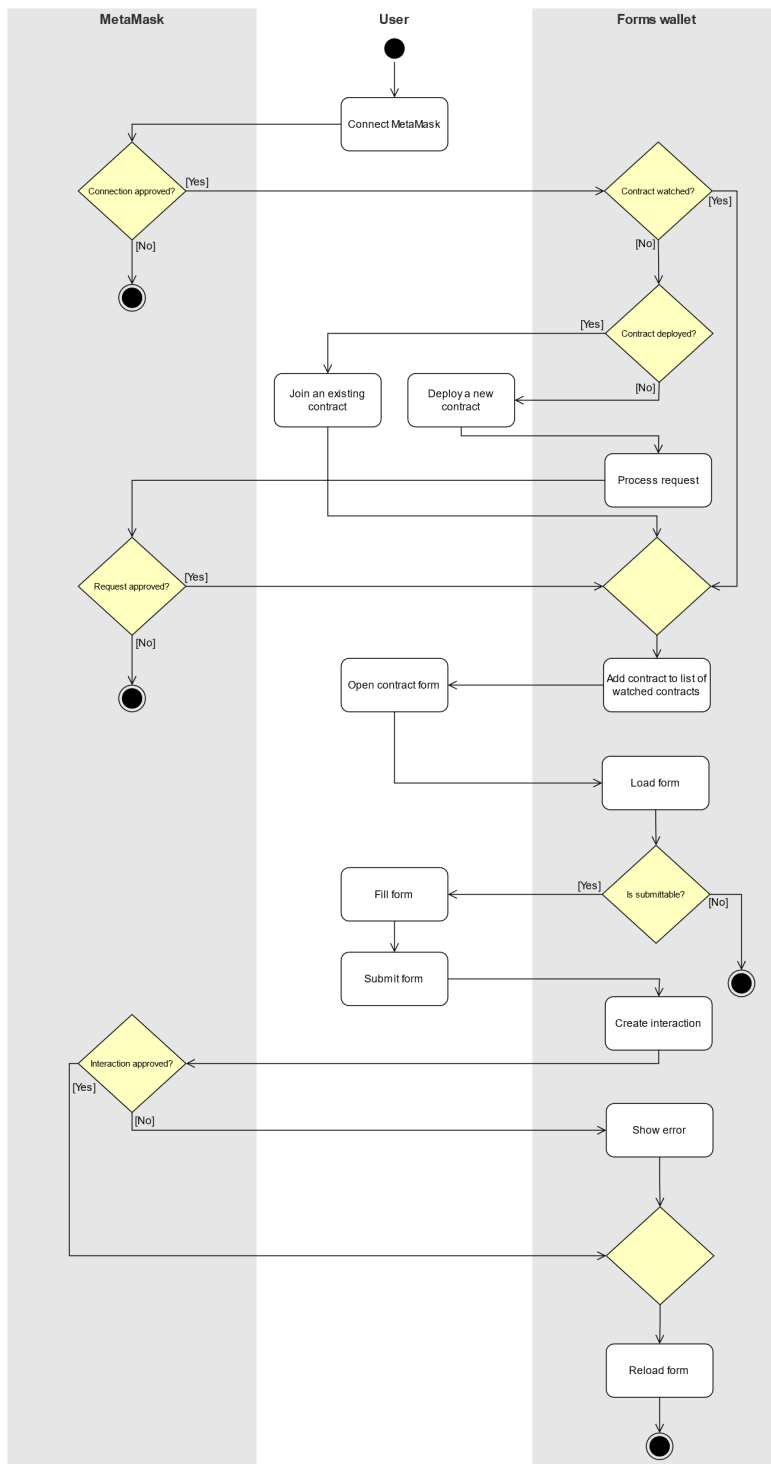


Figure 4.1: Activity diagram of interaction with smart contract

#### 4. PROOF OF CONCEPT

---

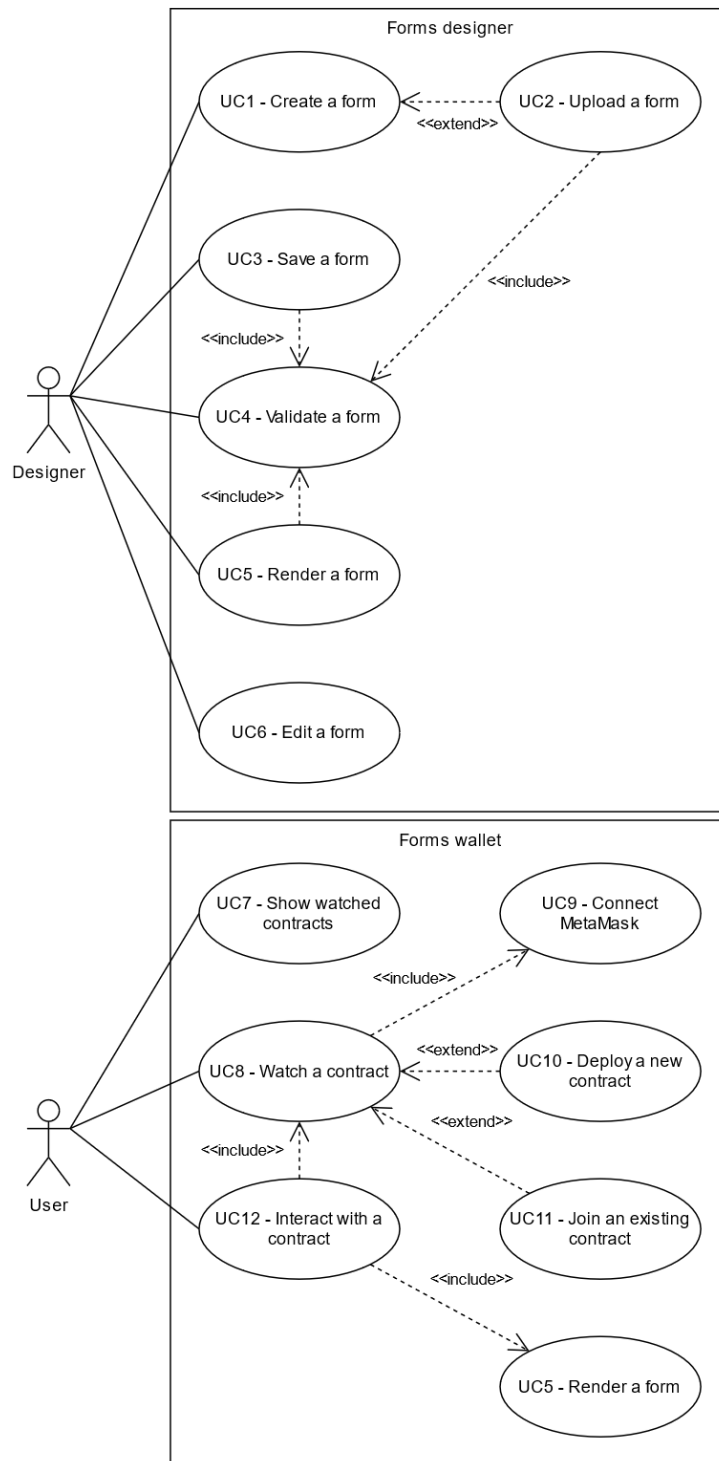


Figure 4.2: Use case diagram for both applications

## 4.3 Use cases

To get more insight into how the forms wallet will be used, figure 4.1 shows how a user can interact with a smart contract within the application. It is also good for identifying use cases for the forms wallet. Figure 4.2 shows use cases for both applications. There are two roles, the user role represents anyone who wishes to interact with smart contracts, and the designer is the person responsible for creating the forms in the forms editor. The use cases are:

- UC1 Create a form** The editor can create an empty form from a default template.
- UC2 Upload a form** The editor can upload an already existing form from local storage instead of using the default template.
- UC3 Save a form** The editor can save the form to local storage.
- UC4 Validate a form** The editor can validate a form against the forms model on demand. Other use cases can trigger validation automatically.
- UC5 Render a form** The editor can preview the form he is currently editing. The form needs to be valid to be rendered. When a user wants to interact with a smart contract, the form needs to be rendered.
- UC6 Edit a form** The editor can edit forms in a user-friendly text editor.
- UC7 Show watched contracts** The user can display all watched smart contracts conveniently.
- UC8 Watch a contract** The user can add a smart contract to interact with to the forms wallet.
- UC9 Connect MetaMask** The user needs to connect the forms wallet to MetaMask to communicate with an Ethereum blockchain.
- UC10 Deploy a new contract** The user can watch a smart contract by deploying it to the Ethereum blockchain. He needs to provide its bytecode, constructor parameters, and a forms XML.
- UC11 Join an existing contract** The user can watch an existing smart contract by providing an Ethereum address and a forms XML.
- UC12 Interact with a contract** The user can interact with a smart contract on an Ethereum blockchain if he is watching it.

## 4.4 Functional and non-functional requirements

We can specify software requirements based on the use cases shown in the previous section and the model specification discussed in the previous chapter. Requirements are classified as functional (F) and non-functional (NF), and out of scope (OS).

### 4.4.1 Forms editor

**F1 Forms model creation** The application has to allow creating the new forms model in the XML syntax (using text editor). The application should make sure that the produced model is valid and can be rendered. If the model is not valid, it should inform the designer in a console why it is not valid. It should also show a preview to the designer for faster designing.

**F2 Exporting and importing the model** The application has to allow saving the created forms model as an XML file. Files created this way can also be loaded into the application for editing or preview. The application should also allow creating new model with basic structure already provided.

**NF1 Text editor usability** The application's text editor has to provide advanced text editing functionalities such as undo, redo, multi-line editing, XML syntax highlighting, and automatic completion.

**NF2 Support for large models** The application should be able to process new forms models of any size. Scrolling in the text editor should be fast and responsive.

**OS1 Live preview** The application will not show live preview of the forms model. Rendering and validation of the model has to be done on-demand.

**OS2 Preview only** The application will only render the visual aspects of the forms model. Any bindings and interaction with the smart contract will be ignored and not processed by the editor.

### 4.4.2 Forms wallet

**F1 Contract watching** The application has to allow the user to watch both generic and Das Contract smart contracts. The application can either deploy the contract on its own if it does not exist yet, or it can watch an Ethereum address where the contract is deployed. The application should ask the user for the forms model associated with the watched contract and any other information required for its deployment or watching.



**F2 Contract listing** All watched contracts and its actions should be available to the user in a menu. The menu should group actions from the same contract together for faster navigation. Additionally, a dashboard should exist that shows all actions available to the user. The dashboard should not group actions based on contract. It should instead allow sorting the actions based on their priority, name or due date. The dashboard should be used as a front page of the application.

**F3 Contract interaction** The application has to allow the user to interact with any smart contract on the Ethereum blockchain that has a correct forms model and is watched by the application. The application should render the form, fill it with data from the blockchain according to the bindings and allow the user to submit this form and interact with smart contracts this way.

**F4 Das Contract flow and roles validation** The application has to only show actions, that are currently available to the user based on his addresses' role and current state of the smart contract. Actions declared in the forms model that are currently not executable on the smart contract should be hidden from the user as they will fail if he tries to execute them.

**NF1 MetaMask integration** The application should handle all blockchain communication through the MetaMask web extension, making it independent on the currently selected Ethereum network.

**NF2 Serverless mode** Once the application is downloaded to the host (usually from a server), it should work even if the server is not available anymore. The application should be decentralized and work as long as it is able to connect to the Ethereum network.

**OS1 Other blockchains** The application will only support Ethereum blockchain. Should the application ever need to work with other blockchains, the data parsing and communication with the network need to be adjusted.

## 4.5 Architecture and design

The whole project consists of two applications (forms editor and forms wallet), which share the forms model and its renderer. Figure 4.3 shows a package diagram of the entire project, how different parts of the applications work on the highest abstraction level, and how they use the forms model.

#### 4. PROOF OF CONCEPT

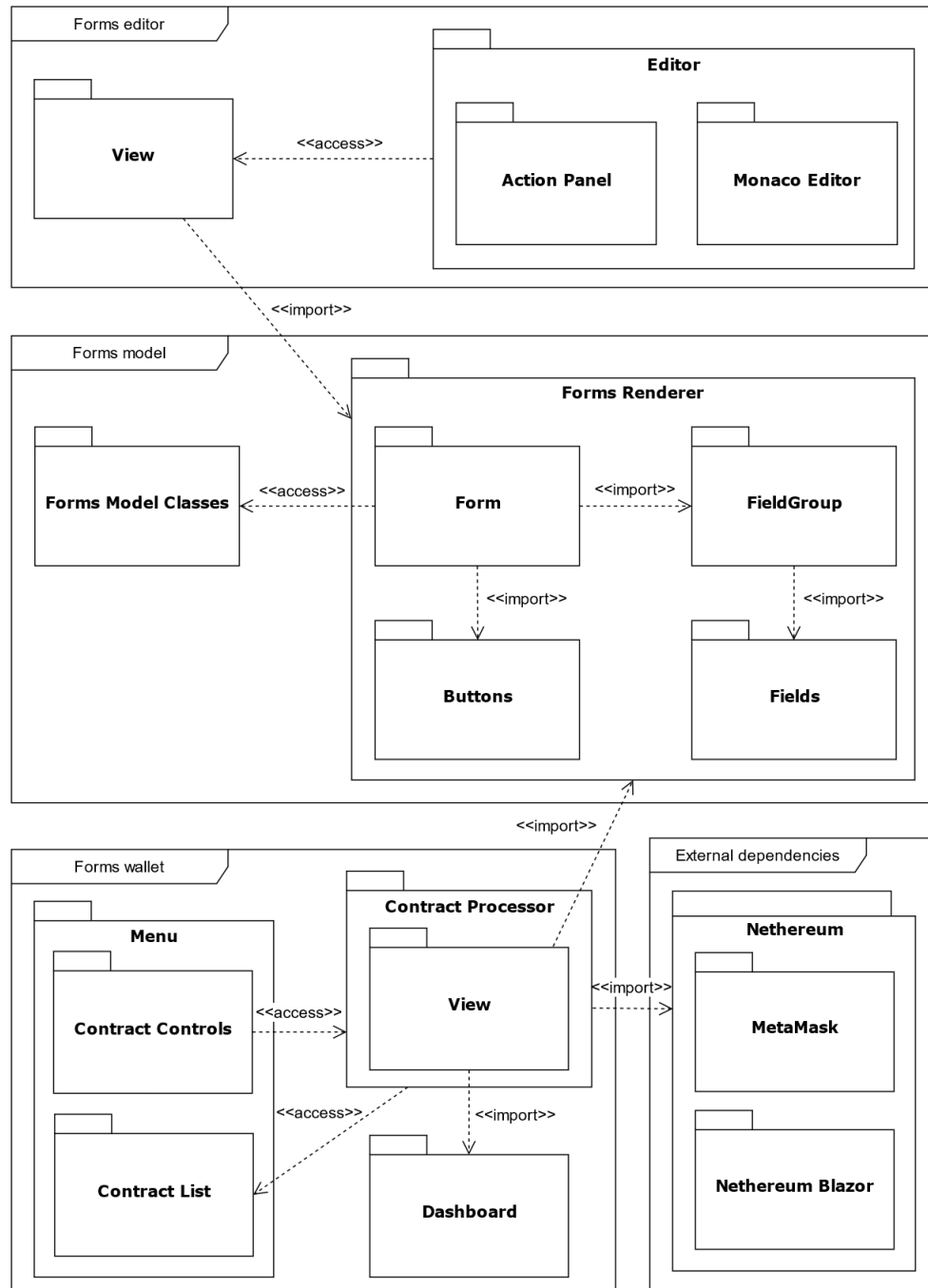


Figure 4.3: Package diagram showing essential parts of the project

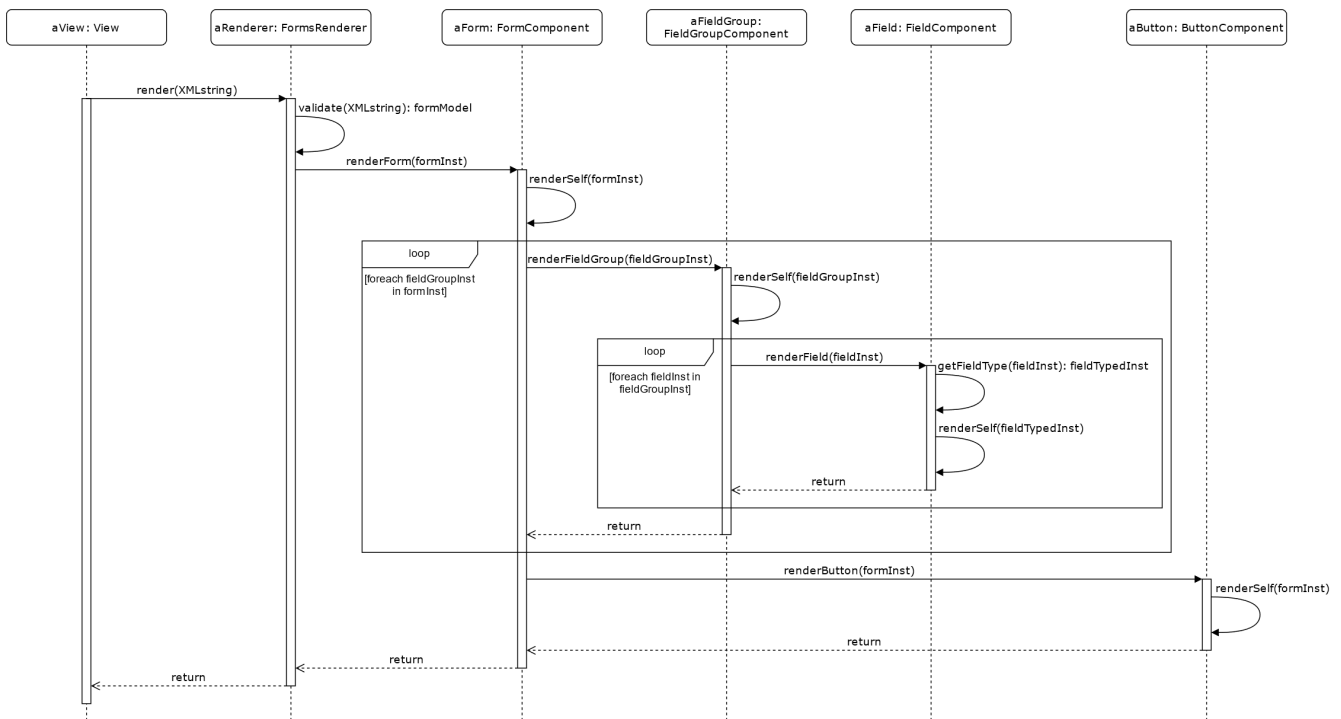


Figure 4.4: Sequence diagram showing how the forms renderer works

### 4.5.1 Forms model

The forms model consists of the forms model classes and the forms renderer. The forms model classes were created from the new Das Contract forms model discussed in the previous chapter and shown in figure 3.11. The forms renderer is a component that can convert the forms model classes into a usable input form in the browser. Figure 4.4 shows a sequence diagram of a view using the forms renderer to create an input form also described below.

The forms renderer creates an instance of the Das Contract forms model and passes it to the Form component. The Form component then renders itself (label, visuals, border) and places a FieldGroup component for each FieldGroup within the forms model instance. It also injects the FieldGroup class from the forms model instance into the FieldGroup component. This approach is then recursive. The FieldGroup component renders itself the same way the Form does and places a Field component for each Field specialization within the forms model instance. The FieldGroup component has to follow the specified Displayed and Vertical rules so that the Fields get rendered correctly. The Field component is then injected with the Field instance from the forms model.

The Field component determines the type of the Field instance and creates

a specific Material Design HTML input element for it, following all parameters specified within the Field instance. In the end, all Field components are two-way data bound to the Fields in the forms model instance. We can iterate over the Form component and access the underlying forms model due to this data binding. Once the recursion calls return to the Forms component, a submit button is created at the bottom, assuming the associated Form instance has a smart contract interaction. The submit button has an on-click event that notifies its subscribers when it was clicked. This subscription is used only in the forms wallet, as the functional requirements specified that the forms editor would not interact with the Ethereum network. Once the forms renderer is finished, it notifies components using it that the form is ready to be used.

### 4.5.2 Forms editor

The forms editor consists of the view and the editor. The editor contains a control panel and an embedded Monaco editor, which is used to create the XML forms. The view imports the forms renderer, allowing it to render the form from any provided Das Contract XML file.

The forms editor is an effortless application consisting of already created components. The only new code is the action panel, which consists of five buttons. The buttons allow the user to create a new model, upload an existing model, save the model, render it and validate it. The action panel propagates these actions to the editor. The editor can access and set the content of the Monaco editor in a string format and call the view to render and validate such a string.

The new model and upload model actions set the content of the editor to the provided string. For a new model, the string is taken from a default template. For an existing model, the string is parsed from the uploaded file. Three actions use the validation of the model: save, render and validate. The validation is as simple as taking a string representing the Das Contract XML and instantiating the forms model with it through the view. If it fails, an exception is thrown, and appropriate steps are taken. If it succeeds, the instantiated forms model can be used for rendering.

The save model action gets the content of the Monaco editor, validates it, and saves it as an XML file. The validation action does the same, except it displays if the model is valid next to the button instead of saving the model. The render action creates the forms model by trying to validate its string representation. Afterward, it tells the view to render it. The view has the rendering functionality imported from the forms renderer described in the previous section. Since this application does not have to interact with the Ethereum network, any events and calls regarding the form are ignored.

### 4.5.3 Forms wallet

The overall architecture of the forms wallet strongly resembles the MVVM design pattern. Chapter 2 discussed the MV\* design patterns for decentralized applications and how we can map the decentralized applications into the MVVM, and it has been applied here. The model is represented by the smart contract on the Ethereum network. The view is represented by the menu and the view components. At last, the view model is represented by the contract processor component and contains all business logic. The menu is used mainly for navigation between contracts and their activities. The contract processor encapsulates a view of the smart contract's form and has the most important functionalities - processing form bindings and communicating with the Ethereum network.

The menu is separated into two components: the contract controls and the contract list. The contract controls contain logic regarding contract watching and contract deployment. They allow the user to deploy a new contract and also join an already deployed contract. If a new contract is supposed to be deployed, the contract controls overlay an input dialog for the contract parameters. Once the user provides all parameters, the contract controls use the contract processor to deploy the contract through MetaMask. If the deployment is successful, the join an existing contract functionality is called and filled automatically. Users can also join an existing contract manually if it is already deployed.

Joining an existing contract is as simple as adding it to the contract list and local storage. The contract is represented by a key-value element, where the key is the contract's Ethereum address and the value is the contract's XML form. The last functionality of the contract controls is the dashboard action. It simply tells the contract processor to switch its view to the dashboard. It will be explained in more detail below.

When the application loads, the contract list loads all contracts from local storage and displays them in a tree view. The tree view has two layers, the first layer being the contract itself and the second layer being the activities the user can currently do within the contract. The contract list also keeps track of the currently displayed smart contract activity (form). If the dashboard is displayed, the currently displayed activity is set to null.

The contract processor contains most of the application's business logic. It contains a dynamic view that shows either a dashboard or the currently selected smart contract activity. The wireframes introduced in figures 3.1, 3.2, 3.3, and 3.4 show how the dynamic view changes between the dashboard and the currently selected smart contract activity in the left tree view menu (when an activity is selected, it is highlighted). The information representing this is stored in the contract list, as explained above. If the dashboard is active, the contract processor retrieves all smart contract activities from the contract list and passes them to the dashboard, displaying them in a list that

can be sorted by their name, due date, and priority.

If a smart contract activity is selected, the contract processor passes the activity's XML form into its view. The view then uses the forms renderer the same way the forms editor does (shown in figure 4.4). However, after the form is done being rendered, the contract processor subscribes to the form's submit button, and it also processes its ViewBinds. These two core functionalities will be explained extensively in the following sections. The communication with the Ethereum network is done through the Nethereum component, which provides Blazor components for connecting to the MetaMask extension and interfaces for different smart contract interactions. Suppose the MetaMask is not connected to the application. In that case, the Nethereum component will automatically overlay the entire application with a connection dialog, making it impossible to use the application without MetaMask connected.

## 4.6 Development process

The previous sections describe how the application works on higher abstraction levels. This section should give us a closer look at some implementation details in the forms wallet application.

### 4.6.1 XML serialization

The problem of converting the forms model into XML format has been mentioned many times in this thesis. In section 3.3, a format with minimal syntax overhead was proposed, which should be used in the implementation. The format was designed to be supported by XML parsers, even though it does not adhere to all formal XML rules. Since we are using Blazor, which supports most .NET libraries, we can use the XMLSerializer from the .NET API.

Figure 4.5 shows some of the forms model classes with XML annotations. By annotating a List of objects as XMLElement instead of XMLArray and XMLArrayItem, we can remove the need for tags surrounding the array items. We are also using the XMLElement to specify that the List of Fields can contain only specializations representing a specific input element. Trying to pass a specialization that does not belong to the annotations or does not inherit from the Field class will fail while serializing. This way, we can rely on the Field interface while also having only elements that can be rendered.

The Currency attribute is ignored in the base class and only annotated in classes that support it in the forms model. This way, the XML format can contain the Currency property only with Fields that support it. The last concept is the Data attribute, which can only be set in runtime through the SetData accessor. The Data attribute is two-way bound into the rendered Material Design component, following the Blazor data binding. When a user changes the data in the input field, it gets propagated directly into the Field

```

public class Form {
    [XmlAttribute("Label")]
    public string Label { get; set; } = "";
    [XmlAttribute("FuncBind")]
    public string FuncBind { get; set; } = "";

    [XmlElement("FieldGroup")]
    public List<FieldGroup> FieldGroups { get; set; } = new List<FieldGroup>();
}

public class FieldGroup {
    [XmlAttribute("Label")]
    public string Label { get; set; } = "";
    [XmlAttribute("Vertical")]
    public bool Vertical { get; set; } = true;
    [XmlAttribute("Displayed")]
    public bool Displayed { get; set; } = true;

    [XmlElement("DateField", typeof(DateField))]
    [XmlElement("AddressField", typeof(AddressField))]
    [XmlElement("SingleLineField", typeof(SingleLineField))]
    [XmlElement("MultiLineField", typeof(MultiLineField))]
    [XmlElement("IntField", typeof(IntField))]
    [XmlElement("DecimalField", typeof(DecimalField))]
    [XmlElement("BoolField", typeof(BoolField))]
    [XmlElement("EnumField", typeof(EnumField))]
    [XmlElement("DropdownField", typeof(DropdownField))]
    public List<Field> Fields { get; set; } = new List<Field>();
}

public abstract class Field {
    [XmlAttribute("ParamBind")]
    public string ParamBind { get; set; } = "";
    [XmlAttribute("ViewBind")]
    public string ViewBind { get; set; } = "";
    [XmlAttribute("Label")]
    public string Label { get; set; }
    [XmlAttribute("Description")]
    public string Description { get; set; } = "";
    [XmlAttribute("ReadOnly")]
    public bool ReadOnly { get; set; } = false;
    [XmlIgnoreAttribute]
    public bool Currency { get; set; } = false;

    public abstract void SetData(string data);
    public abstract object GetData();
}

public class SingleLineField : Field {
    [XmlAttribute("Currency")]
    public new bool Currency { get; set; } = false;

    [XmlIgnoreAttribute]
    public string Data { get; set; }

    public override void SetData(string data) {
        Data = data;
    }

    public override object GetData() {
        return Data;
    }
}

public class BoolField : Field {
    [XmlIgnoreAttribute]
    public bool Data { get; set; }

    public override void SetData(string data) {
        Data = Convert.ToBoolean(data);
    }

    public override object GetData() {
        return Data;
    }
}

```

Figure 4.5: Forms model classes with XML annotations

within the forms model. This allows us to separate the view from the view model and its business logic.

#### 4.6.2 ViewBind data binding

Section 3.6.3 describes the ViewBind format and specifies how to use this format to retrieve information from the blockchain. Whenever the form is rendered within the forms wallet, a function responsible for loading the forms model with data from the blockchain. Since the forms model is data-bound to the rendered input fields, the retrieved values get propagated to the view and shown to the user as soon as the loading is finished.

When the form is finished rendering, the loading function iterates over every field and checks if it has a ViewBind parameter. If it does, it converts

```
class ViewToken {
    public string FuncName { get; set; } = "";
    public List<string> Parameters { get; set; } = new List<string>();
    public string Property { get; set; } = "";
    public int Index { get; set; } = -1;
}
```

Figure 4.6: ViewToken class used to parse data from the blockchain

this parameter from a string into a ViewToken. The ViewToken is shown in figure 4.6. The application uses the Nethereum interface to invoke a call (call on the blockchain is used to get data from it and does not cost any fees). The call requires a function name to be called and its parameters if it accepts any. These are stored in the ViewToken. Suppose such a variable or function exists on the Ethereum blockchain (public variables have a getter with the same name automatically generated). In that case, it is called, and we receive a complex Nethereum structure representing the data received.

The Nethereum structure allows us to check if we can iterate over the data. In case the data are iterable, and the index is in bounds and specified, we use the string representation of the data at the provided index. If the index is not specified, we use either the first value or the entire array based on the field class (MultiLineField and ComboBoxField types use the whole collection). If the data is structured, we use the Property from the ViewToken to get the correct attribute to put into the input field. Suppose the designer makes an invalid ViewBind that our application cannot interpret or the data do not exist on the blockchain. In that case, it is silently ignored because, in most cases, being unable to see a specific field from the blockchain will not stop the user from completing the activity.

### 4.6.3 ParamBind data binding

The ParamBind attribute specifies which inputs should be used to execute a smart contract transaction. Suppose we have a Das Contract activity bound to our form, and it takes parameters  $a$ ,  $b$ , and  $c$  to execute it (in a generic smart contract, this is equivalent to calling transaction  $func(a, b, c)$ ). When the user submits the form, we need to iterate over all Fields within the forms model instance and retrieve parameters  $a$ ,  $b$ , and  $c$ . For this purpose, we use the ParamBind attribute. Therefore, we are looking for fields whose ParamBind is set to  $a$ ,  $b$ , and  $c$ , respectively. Once we find these fields, we can use the GetData method to get an untyped object. The Nethereum interface for calling transactions unknown at compile time takes an array of objects as an argument to call such transaction. We just need to make sure that the order of the parameters matches the transaction call. The Nethereum interface



then converts these objects into the correct format, calculates the associated transaction fees, and sends the transaction to the MetaMask extension, where the user has to confirm it. Using ParamBind allows us to interact with both Das Contract and generic smart contracts. It also allows the designer to order the input fields however he wants to, thus allowing him to make a more user-friendly interface.

## 4.7 Testing

The project uses bUnit for conducting unit tests. bUnit is a testing library for Blazor Components. It allows both code and component testing and interaction with the components and mockup of common Blazor services.[48] Most of the focus was put on the ViewBind parsing in the unit tests as it is most prone to errors. The unit tests make sure that the syntax is fully supported and gets parsed into tokens correctly. Besides that, another functionality being used extensively throughout the application is the validation of Ethereum addresses. This functionality is used primarily when an existing contract is added by the user and in other cases where we work with Ethereum addresses.

A lot of the critical functionalities are executed by external components. For example, the XML serialization uses the standard .NET library, making sure that the conversion is always correct. The Nethereum library provides a communication interface with the blockchain and MetaMask integration. Using the Nethereum library solves many problems, such as converting data types to blockchain-specific format and parsing raw values. Those functionalities would typically have to be tested, but since the projects are popular and established, our test would probably never find a new bug if it existed.

The project functionality and smart contract interaction were tested manually on the Ganache blockchain. Ganache is a personal Ethereum blockchain made for rapid DApp development.[49] The results will be presented in the following section. We were able to execute the entire Das Contract mortgage contract, including interaction with smart contracts outside of the Das Contract project. Afterward, the same steps were reproduced on the Ropsten testnet, which closely resembles the main Ethereum blockchain.

## 4.8 Project showcase - a decentralized mortgage contract

Both applications are released under the MIT license on Github<sup>3,4</sup> and also deployed on Azure<sup>5,6</sup>. To demonstrate the functionality of the project, we will

---

<sup>3</sup>Forms wallet source code: <https://github.com/ancinpet/thesis-wallet>

<sup>4</sup>Forms editor source code: <https://github.com/ancinpet/thesis-forms-editor>

<sup>5</sup>Forms wallet release: <https://witty-bay-09186f103.azurestaticapps.net>

<sup>6</sup>Forms editor release: <https://yellow-beach-0c4abc503.azurestaticapps.net>

## 4. PROOF OF CONCEPT

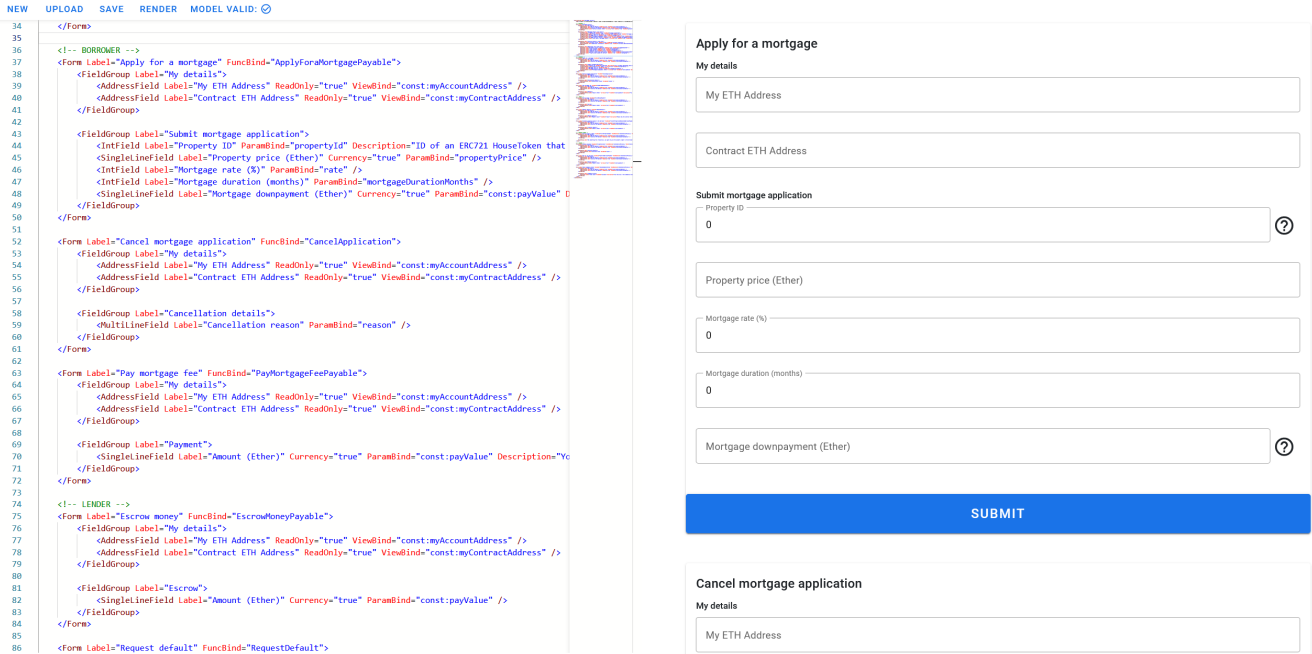


Figure 4.7: Final look of the forms editor

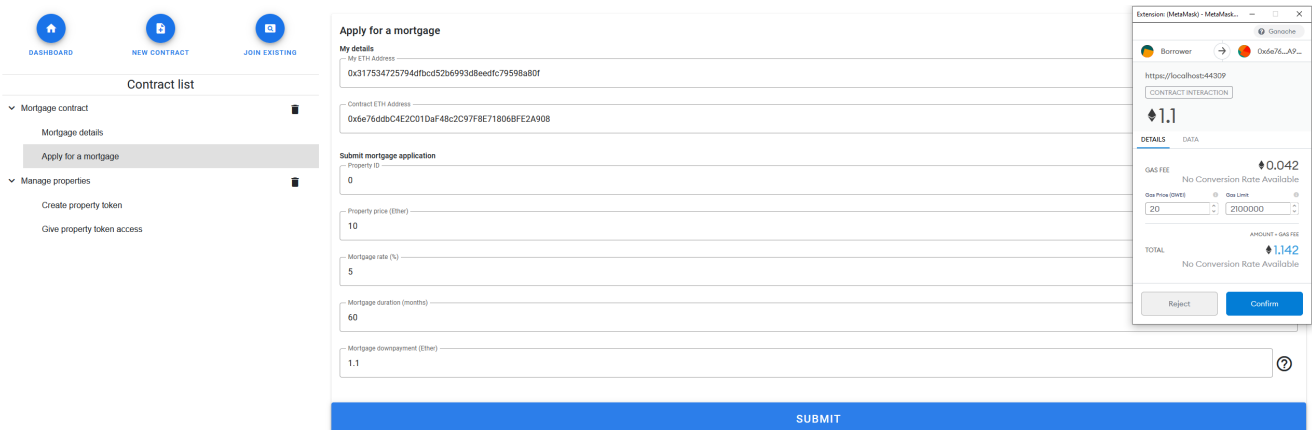


Figure 4.8: Final look of the forms wallet while interacting with Das Contract

## 4.8. Project showcase - a decentralized mortgage contract

The screenshot shows a web application interface with a navigation menu on the left containing 'DASHBOARD', 'NEW CONTRACT', and 'JOIN EXISTING'. The main content area is titled 'Contract list' and has a sidebar with options: 'Manage properties' (with sub-options 'Create property token' and 'Give property token access'), and 'Mortgage contract' (with sub-options 'Mortgage details' and 'Escrow property rights'). The 'Give property token access' form is active, featuring two sections: 'My details' with 'My ETH Address' (0x0bd3bd11d8bb5a88b6849f837acd120c7395e) and 'Contract ETH Address' (0x15a0876600A6C2A93c2eb6EECF94e25484461c4); and 'Give token rights to a contract' with 'Contract address' (0x6e76ddb0c4E2C01DaF48c2C97F8E718068FE2A908) and 'Token ID' (0). A blue 'SUBMIT' button is at the bottom. An Ethereum wallet extension window is open on the right, showing a 'CONTRACT INTERACTION' for 'Property OL...' with a gas fee of 0.042 ETH and a gas limit of 200,000. The window has 'Reject' and 'Confirm' buttons.

Figure 4.9: Final look of the forms wallet while interacting with generic smart contract

The screenshot shows the 'Mortgage details' form in the same wallet interface. The sidebar now highlights 'Mortgage details' and 'Play mortgage fee'. The form is divided into several sections: 'My details' with 'My ETH Address' (0x317534725794dfbcd52b6993d8eedfc79598a80f) and 'Contract ETH Address' (0x6e76ddb0c4E2C01DaF48c2C97F8E718068FE2A908); 'Contract balance (Ether)' (0); 'Mortgage details (view only)' with 'Property ID' (0), 'Property price (Ether)' (10), 'Mortgage rate (%)' (5), 'Mortgage downpayment (Ether)' (1.1), 'Mortgage duration (months)' (60), and 'Total paid on mortgage (Ether)' (0); 'Insurance details (view only)' with a checked 'Insurance active' checkbox and a text area containing 'I accept'; and 'Additional info (view only)' with an unchecked 'Mortgage contract completed' checkbox.

Figure 4.10: Final look of the forms wallet while showing Das Contract details

use the Das Contract mortgage contract introduced in section 1.11.1 and figure 1.9. The mortgage contract allows a user to take out a mortgage without any central authority. In our scenario, the mortgage contract will use an ERC-721 house token explicitly created for testing the contract.

The forms model has to be designed in the forms editor to interact with both smart contracts. Figure 4.7 shows the final look of the forms editor while modeling the mortgage contract in it. The resulting forms model is released on Github<sup>7</sup>. The Source folder contains Solidity source codes of both contracts. The Forms folder contains the forms model for interacting with them, and the Build folder contains compiled sources to provide bytecode for contract deployment.

The following YouTube video<sup>8</sup> shows a full walkthrough of the Das Contract mortgage contract, including interaction with the house token contract, which is not part of the Das Contract project. The video can also be found on the enclosed DVD. In the walkthrough, we use the Ganache blockchain over a live testnet blockchain so that we do not have to wait for transactions to be mined. Figures 4.8, 4.9, and 4.10 show the wallet's final look while interacting with both generic and Das Contract smart contracts.

---

<sup>7</sup><https://github.com/ancinpet/thesis-wallet/tree/main/src/thesis-wallet/Contracts>

<sup>8</sup><https://www.youtube.com/watch?v=Z3dTFiMwZTU>

---

# Conclusion

The goal of this thesis was to propose a domain-specific language that would allow defining decentralized application's user interfaces compatible with the Das Contract language without much effort.

Chapter 1 provided the essential blockchain and Das Contract knowledge required to propose such domain-specific language. In chapter 2, the standard and approaches to declarative user interface were reviewed. Chapter 2 revealed that the MVVM design pattern is the most suitable one from the MV\* pattern family when it comes to decentralized applications. It also revealed two possible approaches for declaring user interfaces in a standardized way. The first approach uses the Interaction Flow Modeling Language, an OMG standard for describing user interface structure and behavior. The second approach consists of using XML-based domain-specific language, which has been used extensively to describe user interfaces, for example, XAML in the .NET framework.

Chapter 3 examined both approaches and tried using them to describe the theoretical user interface of our decentralized application. Based on this examination, we were able to conclude that the XML-based domain-specific language is more suitable for our use case, and we were able to propose a new forms model for the Das Contract language. The new forms model uses modified XML syntax with minimal overhead. It allows describing user interfaces that can interact with both Das Contract and generic smart contracts. It is relatively easy to understand and design in at first sight because it closely resembles HTML. At the end of chapter 3, we designed the entire Das Contract mortgage contract in our new proposed forms model.

During this thesis, two applications were created and described in chapter 4. The forms editor is an application for designing user interfaces in the new domain-specific language. It consists of a fully-featured code editor and a forms model renderer and validator, making the designing process easier. The second application is the forms wallet. It is a proof of concept implementation of the forms model interpret used to interact with both Das Contract and

generic smart contracts on the Ethereum network. The end of chapter 4 showcases the entire project, including a complete walkthrough of the Das Contract mortgage contract.

In the future, the forms model could be extended to allow the designer to create forms with multiple steps, either through pagination or by displaying new fields gradually as previous sections are filled. The forms editor could be extended to validate forms in real-time against an XML scheme. Additionally, the forms editor could implement a custom IntelliSense, which would create complete fields with a single button press and offer better code completion. The forms wallet could be extended to add implementation for other blockchains, and it could also automatically create a forms model from a smart contract's source or an ABI.

---

## Bibliography

1. NAKAMOTO, Satoshi. *Bitcoin: A Peer-to-Peer Electronic Cash System* [online]. 2008 [visited on 2021-02-19]. Tech. rep. Available from: <https://bitcoin.org/bitcoin.pdf>.
2. BASHIR, Imran. *Mastering Blockchain: Distributed ledger technology, decentralization, and smart contracts explained*. 2nd ed. Packt Publishing, 2018. ISBN 978-1788839044.
3. GATES, Mark. *Blockchain: Ultimate guide to understanding blockchain, bitcoin, cryptocurrencies, smart contracts and the future of money*. CreateSpace Independent Publishing Platform, 2017. ISBN 978-1547090686.
4. NASCIMENTO, S.; PÓLVORA, A.; ANDERBERG, A.; ANDONOVA, E.; BELLIA, M.; CALÈS, L.; INAMORATO DOS SANTOS, A.; KOUNELIS, I.; NAI FOVINO, I.; PETRACCO GIUDICI, M.; PAPANAGIOTOU, E.; SOBOLEWSKI, M.; ROSSETTI, F.; SPIRITO, L. *Blockchain Now And Tomorrow: Assessing Multidimensional Impacts of Distributed Ledger Technologies*. Luxembourg: Publications Office of the European Union, 2019. ISBN 978-92-76-08977-3.
5. DICKMAN, Tom. *Crypto51* [online]. 2020 [visited on 2020-12-02]. Available from: <https://www.crypto51.app/>.
6. CAMBRIDGE. *Bitcoin Mining Map* [online]. 2021 [visited on 2021-02-01]. Available from: [https://cbeci.org/mining\\_map](https://cbeci.org/mining_map).
7. ANTONOPOULOS, Andreas. *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, Inc, 2018. ISBN 978-1491971949.
8. PALLADINO, Santiago. *Ethereum for Web Developers: Learn to Build Web Applications on top of the Ethereum Blockchain*. Apress, 2019. ISBN 978-1-4842-5278-9.
9. INC., ConsenSys Software. *A crypto wallet & gateway to blockchain apps* [online] [visited on 2021-02-02]. Available from: <https://metamask.io/>.

## BIBLIOGRAPHY

---

10. HUSSEY, Matt; PHILLIPS, Daniel. *MetaMask: What It Is and How To Use It* [online] [visited on 2021-02-02]. Available from: <https://decrypt.co/resources/metamask>.
11. SZABO, Nick. Formalizing and Securing Relationships on Public Networks. *First Monday*. 1997, vol. 2, no. 9. Available from DOI: 10.5210/fm.v2i9.548.
12. BASHIR, Imran. *Mastering Blockchain: A deep dive into distributed ledgers, consensus protocols, smart contracts, DApps, cryptocurrencies, Ethereum, and more*. 3rd ed. Packt Publishing, 2020. ISBN 978-1-83921-319-9.
13. SWAN, Melanie. *Blockchain: Blueprint for a New Economy*. O'Reilly, 2015. ISBN 978-1491920497.
14. VOGELSTELLER, Fabian; BUTERIN, Vitalik. *EIP-20: ERC-20 Token Standard* [online]. 2015 [visited on 2021-02-25]. Available from: <https://eips.ethereum.org/EIPS/eip-20>.
15. ENTRIKEN, William; SHIRLEY, Dieter; EVANS, Jacob; SACHS, Nastassia. *ERC-721 Non-Fungible Token Standard* [online]. 2018 [visited on 2021-02-25]. Available from: <https://eips.ethereum.org/EIPS/eip-721>.
16. ELLIS, Steve; JUELS, Ari; NAZAROV, Sergey. *ChainLink: A Decentralized Oracle Network* [online]. 2017-09 [visited on 2021-02-24]. Tech. rep. Available from: <https://link.smartcontract.com/whitepaper>.
17. OPENBAZAAR. *TRULY DECENTRALIZED, PEER-TO-PEER ECOMMERCE* [online] [visited on 2021-02-02]. Available from: <https://openbazaar.org/features/>.
18. LAZOOZ. *A value system designed for sustainability* [online] [visited on 2021-02-02]. Available from: <http://lazooz.org/>.
19. CORP., Eva Global. *Coop ridesharing* [online] [visited on 2021-02-02]. Available from: <https://eva.coop/>.
20. *DRIFE - Taxi 3.0* [online] [visited on 2021-02-02]. Tech. rep. DRIFE Technologies. Available from: <https://www.drife.one/docs/DRIFE-WhitePaper.pdf>.
21. TWISTER. *Peer-to-peer microblogging* [online] [visited on 2021-02-02]. Available from: <http://twister.net.co/about/>.
22. *The Maker Protocol: MakerDAO's Multi-Collateral Dai (MCD) System* [online] [visited on 2021-02-02]. Tech. rep. Maker Community. Available from: <https://makerdao.com/en/whitepaper/>.
23. INC., Dapper Labs. *CryptoKitties* [online] [visited on 2021-02-02]. Available from: <https://www.cryptokitties.co/>.



24. FOUNDATION, The Decentraland. *Decentraland DAO - The virtual world in your hands* [online] [visited on 2021-02-02]. Available from: <https://dao.decentraland.org/en/>.
25. SKOTNICA, Marek; PERGL, Robert. Das Contract - A Visual Domain Specific Language for Modeling Blockchain Smart Contracts. In: *Advances in Enterprise Engineering XIII*. Cham: Springer International Publishing, 2020, pp. 149–166. ISBN 978-3-030-37933-9.
26. SKOTNICA, Marek; KLICPERA, Jan; PERGL, Robert. Towards Model-Driven Smart Contract Systems – Code Generation and Improving Expressivity of Smart Contract Modeling. In: *Proceedings of the 20th CIAO! Doctoral Consortium, and Enterprise Engineering Working Conference Forum 2020 co-located with 10th Enterprise Engineering Working Conference (EEWC 2020)* [online]. CEUR Workshop Proceedings (CEUR-WS.org), 2020 [visited on 2021-04-17]. Available from: <http://ceur-ws.org/Vol-2825/paper1.pdf>.
27. SKOTNICA, Marek. *Das Contract mortgage case study* [online]. 2020 [visited on 2021-03-27]. Available from: <https://github.com/CCMiResearch/DasContract/tree/master/DasContract.CaseStudies/mortgage>.
28. SKOTNICA., Marek; APARÍCIO., Marta; PERGL., Robert; GUERREIRO., Sérgio. Process Digitalization using Blockchain: EU Parliament Elections Case Study. In: *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD*, SciTePress, 2021, pp. 65–75. ISBN 978-989-758-487-9. Available from DOI: 10.5220/0010229000650075.
29. KOURAKLIS, John. *MVVM in Delphi: Architecting and Building Model View ViewModel Applications*. 1st ed. Apress, 2016. ISBN 978-1-4842-2214-0.
30. FOWLER, Martin. *Patterns of Enterprise Application Architecture*. 1st ed. Addison-Wesley Professional, 2002. Addison-Wesley. ISBN 978-0-3211-2742-6.
31. FOWLER, Martin. *GUI Architectures* [online]. 2006 [visited on 2021-03-13]. Available from: <https://martinfowler.com/eaDev/uiArchs.html>.
32. BUSCHMANN, Frank; MEUNIER, Regine; ROHNERT, Hans; SOMMERLAD, Peter; STAL, Michael. *Pattern-oriented software architecture*. Wiley, 1996. ISBN 978-0-471-95869-7.
33. HALL, Gary. *Pro WPF and Silverlight MVVM: Effective Application Development with Model-View-ViewModel*. 1st ed. Apress, 2010. Expert’s Voice in WPF. ISBN 978-1-4302-3162-2.
34. *The MVVM Pattern* [online]. 2012 [visited on 2021-03-13]. Available from: <https://msdn.microsoft.com/en-us/library/hh848246.aspx>.

35. MILLER, Robert C. *Lecture 9: Declarative UI*. 2006. Available also from: <http://courses.csail.mit.edu/6.831/archive/2006/lectures/L9.pdf>.
36. FOWLER, Martin. *Domain-Specific Languages*. Addison-Wesley Professional, 2010. ISBN 978-0-1321-0754-9.
37. *Xamarin.Forms XAML Basics* [online]. 2017 [visited on 2021-03-21]. Available from: <https://docs.microsoft.com/en-us/xamarin/xamarin-forms/xaml/xaml-basics/>.
38. *Interaction Flow Modeling Language* [online]. 2015 [visited on 2021-03-22]. Available from: <https://www.omg.org/spec/IFML/1.0/PDF>.
39. BRAMBILLA, Marco; FRATERNALI, Piero. *Interaction Flow Modeling Language: Model-Driven UI Engineering of Web and Mobile Apps with IFML*. Elsevier, 2015. ISBN 978-0-1280-0108-0.
40. BERNASCHINA, Carlo. *IFMLEdit.org - A web based tool for prototyping and developing web and mobile apps*. 2020. Available also from: <https://www.ifmledit.org/>.
41. *Blazor: Build client web apps with C#: .NET* [online]. Microsoft [visited on 2021-04-24]. Available from: <https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor>.
42. TOMASSETTI, Gabriele. *Blazor: .NET in the Browser* [online]. Strumenta, 2021-01 [visited on 2021-04-24]. Available from: <https://tomassetti.me/blazor-net-in-the-browser/>.
43. BLANCO, Juan. *What is Nethereum* [online]. Nethereum Documentation [visited on 2021-04-24]. Available from: <http://docs.nethereum.com/en/latest/>.
44. BLANCO, Juan. *Nethereum Metamask Blazor* [online]. GitHub [visited on 2021-04-24]. Available from: <https://github.com/Nethereum/Nethereum.Metamask.Blazor>.
45. *About Monaco Editor* [online]. Microsoft [visited on 2021-04-24]. Available from: <https://microsoft.github.io/monaco-editor/>.
46. *About Material Design* [online]. Google [visited on 2021-04-24]. Available from: <https://material.io/design/introduction#principles>.
47. SAMOILENKO, Vladimir. *MatBlazor* [online]. GitHub [visited on 2021-04-24]. Available from: <https://www.matblazor.com/>.
48. HANSEN, Egil. *bUnit* [online]. GitHub [visited on 2021-04-25]. Available from: <https://github.com/bUnit-dev/bUnit>.
49. *Ganache* [online]. EgilConsenSys Software Inc. 2021 [visited on 2021-04-25]. Available from: <https://www.trufflesuite.com/ganache>.

## Acronyms

<b>ABI</b>	Application Binary Interface
<b>BPMN</b>	Business Process Model and Notation
<b>DApp</b>	Decentralized Application
<b>DOM</b>	Document Object Model
<b>DSL</b>	Domain-Specific Language
<b>ETH</b>	Ether
<b>EVM</b>	Ethereum Virtual Machine
<b>IFML</b>	Interaction Flow Modeling Language
<b>MVC</b>	Model–View–Controller
<b>MVP</b>	Model–View–Presenter
<b>MVVM</b>	Model–View–Viewmodel
<b>NFT</b>	Non-Fungible Token
<b>OMG</b>	Object Management Group
<b>POW</b>	Proof of Work
<b>UI</b>	User Interface
<b>UML</b>	Unified Modeling Language
<b>XML</b>	Extensible Markup Language
<b>XAML</b>	Extensible Application Markup Language



---

## Contents of enclosed CD

readme.txt .....	the file with CD contents description
src .....	the directory of source codes
├─ wallet .....	implementation sources of the forms wallet
├─ editor .....	implementation sources of the forms editor
└─ thesis .....	the directory of L <sup>A</sup> T <sub>E</sub> X source codes of the thesis
text .....	the thesis text directory
└─ thesis.pdf .....	the thesis text in PDF format
other .....	other thesis resources
└─ showcase.mp4 .....	video showcasing the forms wallet application