

Article

Smart Contract Design Pattern for Processing Logically Coherent Transaction Types

Tomasz Górski 

Institute of Computer Science, University of Gdańsk, Wita Stwosza 57, 80-308 Gdańsk, Poland;
tomasz.gorski@ug.edu.pl

Abstract: Recent research shows that the source code of smart contracts is often cloned. The processing of related types of transactions in blockchain networks results in the implementation of many similar smart contracts. The rules verifying transactions are therefore duplicated many times. The article introduces the *AdapT* v2.0 smart contract design pattern. The design pattern employs a distinct configuration for each transaction type, and verification rule objects are shared among configurations. The redundancy of logical conditions was eliminated at two levels. Firstly, it is possible to combine similar smart contracts into one. Secondly, a configuration in a smart contract reuses verification rule objects at runtime. As a result, only one object is instantiated for each verification rule. It allows for the effective use of operating memory by the smart contract. The article presents the implementation of the pattern using object-oriented and functional programming mechanisms. Applying the pattern ensures the self-adaptability of a smart contract to any number of transaction types. The performance tests were carried out for various numbers of verification rules in a smart contract and a different number of checked transactions. The obtained evaluation time of 10,000,000 transactions is less than 0.25 s.

Keywords: blockchain; smart contract; object-oriented programming; functional programming; design pattern



Citation: Górski, T. Smart Contract Design Pattern for Processing Logically Coherent Transaction Types. *Appl. Sci.* **2024**, *14*, 2224. <https://doi.org/10.3390/app14062224>

Academic Editors: Chilukuri K. Mohan and Gianluca Lax

Received: 7 February 2024

Revised: 21 February 2024

Accepted: 6 March 2024

Published: 7 March 2024



Copyright: © 2024 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Smart contracts are software that controls the execution of transactions in blockchain networks. An overview of smart contracts was presented by Zheng et al. [1] with a discussion on the challenges and recent technical advances. They reveal that smart contracts are written mostly in the following programming languages: Solidity, Go, Java, and Kotlin. They also indicate problems that plague current blockchain platforms, i.e., re-entrance, block randomness, and overcharging. They emphasize that it stems from the under-optimization of smart contract source code, in which various anti-patterns may be found (e.g., dead code or expensive operations in loops consisting of repeated computations). Blockchain technology is increasingly used in a wide range of applications. Hence, the topic of designing, programming, and testing smart contracts is becoming more and more important. Lately, Wu et al. [2] reviewed the progress that has been made in smart contracts. They confirmed the essence of the design process, and showed the smart contract life cycle including the phases: contract generation, contract release, and contract execution. The authors pointed out the main problems in the development of smart contracts in the areas of performance, privacy, and security. As for efficiency, they underlined that the problem lies in low contract execution efficiency. Moreover, Kannengießner et al. [3] identified challenges in smart contract design, proposed solutions, and recommended software design patterns. Their recommendations sometimes refer to existing software design patterns previously proposed by Gamma et al. [4], e.g., *Proxy Pattern* and *Facade Pattern*. However, they also show smart contract-specific patterns. Researchers and practitioners propose using the *Oracle Pattern* whenever external data are required by a smart contract. Additionally, the known

threat in smart contracts is a reentrancy attack. The authors propose using *Mutex Pattern* or *Checks-Effects-Interactions Pattern* to eliminate that smart contract vulnerability. In the area of efficiency, Six et al. [5] pointed out patterns that enhance execution, storage, and redundancy aspects, e.g., *Incentive Execution*, *Limit Storage*, and *Avoid Redundant Operations*. In addition, researchers work on different ways to execute smart contracts on blockchain networks. Recently, Liu et al. [6] proposed parallel processing of transactions that increased the level of throughput. Design patterns are constantly evolving. Gupta et al. [7] proposed Proxy Smart Contracts that serve as intermediaries in the execution of the actual smart contracts. That pattern is used for the communication of on-chain smart contracts with off-chain services. Another software design pattern that has been employed for blockchain smart contracts is the *Delegation Pattern*. Kim et al. [8] have applied it to the construction of updatable smart contracts to provide federated authentication schemes. Additionally, *Proxy* and *Delegation* patterns are typically used in the upgrading mechanism of Ethereum smart contracts [9]. A different manner of standardizing the design of smart contracts is applying templates. Chu et al. [10] review works that not only identify vulnerabilities, but also offer mechanisms and tools to eliminate them. They show the currently developed mechanisms used to repair vulnerabilities of smart contracts operating in off-chain and on-chain modes.

However, the currently developed design patterns for smart contracts do not cover the subject of their reconfiguration, in particular the possibility of adapting to various types of processed transactions. A smart contract can perform the same operation for logically consistent but different types of transactions. That may include, e.g., on-chain and off-chain transactions, authentication and authorization in the system for different roles, in-community and cross-community energy transfers, and domestic and foreign contract management for the cross-border labor market. Hence, a need to define a smart contract design pattern that would enable operations on various types of transactions. Górski presented the design pattern for reconfigurable smart contracts in [11]. However, it only allowed operation on two types of transactions. In addition, the reconfiguration of the smart contract entailed the need to create new objects of verification rule classes. As a result, it engaged the garbage collector every time the transaction type changed. Additionally, the abstract layer of the pattern was not entirely independent of the specific smart contract.

Figure 1 illustrates smart contract configurations for transaction types, which share a set of unique verification rule objects. The following symbols are used in the figure: obj_i means the object of the i -th verification rule and ref_i means a reference to the obj_i . Verification rule objects are shared within one smart contract, which allows checking logically related transactions.

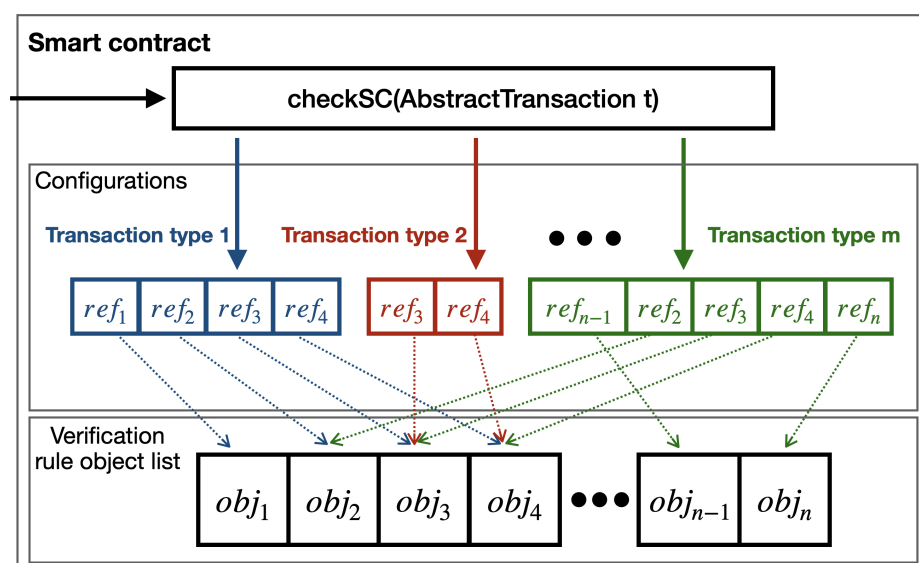


Figure 1. Configurations share verification rule objects from the same list.

The main contributions of this paper are listed as follows:

- The *AdapT* v2.0 design pattern that allows processing any number of transaction types;
- A redesign of the abstract layer of the pattern;
- An implementation of the abstract layer in Java language, which employs object-oriented and functional programming. The implementation of the abstract layer is independent of the specific smart contract;
- Implementation of the concrete layer of the pattern in Java language on the example of a smart contract for energy transmission in prosumer communities;
- Reuse and redundancy metrics with analysis of the pattern;
- Performance tests of the pattern for the number of transactions ranging from 100,000 to 10,000,000.

The remaining part of this paper has the following structure. Section 2 presents related studies on the topic tackled in the paper. Section 3 introduces the design of the *AdapT* pattern. The section also contains the implementation of the pattern in Java. In Section 4, an analysis of the re-use and redundancy was enclosed, whereas Section 5 depicts the results of performance tests. Section 6 contains a discussion on the pros and cons of the pattern. Section 7 summarizes the work performed and lists the planned future tasks.

2. Related Work

The topic of smart contract development is a very fast-growing branch of software engineering. Despite the fact that significant progress has been made in the area of smart contract design, many challenges remain. The interview among professionals performed by Zou et al. [12] reveals the following obstacles that developers must face: lack of design methods of secure smart contracts, basic existing development tools, limitations of programming languages, performance constraints, and limited online resources. Vacca et al. [13] investigated tools developed for smart contracts. They highlight that the majority of the tools target the Ethereum framework, e.g., *SolMet*, *GasChecker*, and *SmartEmbed*. Only a few tools were written for Hyperledger Fabric, e.g., *Zeus* and *Blockbench*. The tools gathered in the review help mainly detect security vulnerabilities in smart contracts written in Solidity language. Two of them have functionality close to software design issues: *SmartEmbed* identifies code clones and *Gasper* locates gas-costly programming anti-patterns. Gas waste in smart contract loops is considered by Li et al. [14] focusing on applying machine learning to detect the *Expensive Operation* anti-pattern. However, none of them verifies the source code of smart contracts on compliance with design patterns. Two design patterns were introduced by Mandarino et al. [15]. The first is an architecture that reduces gas consumption in case of updating the source code of a smart contract. The second shows smart storing of data in the form of packing bits representing boolean values. The pattern for communication of on-chain smart contracts with off-chain services was proposed by Liu et al. [16]. They proposed a data carrier architecture that consists of three components: Mission Manager, Task Publisher, and Worker. Those components interact with smart contracts and off-chain data sources.

Researchers also propose templates as an alternative way to unify smart contract design. For example, Jin et al. [17] developed the *Aroc* tool, which generates a patch contract containing security rules based on the fixed template and deploys it to the blockchain. Templates are also the subject of other research. Furthermore, Gec et al. [18] propose a support system that recommends and provides smart contract source code templates suitable for a fog architecture, whereas Mao et al. [19] operate at a higher level of abstraction. They provide a set of specialized templates of basic functions for users to design smart contracts visually by the user interface.

An interesting area where design patterns may appear is model-driven engineering, because generating smart contract source codes requires some template or design pattern. In this context, Bodorik et al. [20] show the transformation that generates the source code of smart contracts from the description of business models presented in Business Process Model and Notation (BPMN). Similarly, Shen et al. [21] also use BPMN to visualize the

process to be understood by the stakeholders from various domains. They introduced a smart contract generator for developing multi-party interaction scenarios. In contrast, Jurgelaitis et al. [22] step lower at the level of system modeling and show the mechanism of generating smart contracts for the Solidity language from Unified Modeling Language (UML) state diagrams.

Currently conducted research works concern the application of blockchain technology and smart contracts in various domains. Solutions for the medical sector are among the most intensively researched (Yang et al. [23]). Additionally, smart contracts are increasingly used in the energy sector, especially in distributed renewable energy systems (Honari et al. [24]). From its beginning, blockchain has been used in the financial sector (Wang et al. [25]). On the other hand, smart contracts are increasingly used in public services such as online voting systems (Saim et al. [26]). Logistics, in particular supply chain management, is also a constantly developed area of application (Natanelov et al. [27]). Interesting research works can be expected in the area of smart contract unification. There is a place for research on both domain-specific and domain-independent design patterns. Anyway, the first papers on these issues have already been published. Wohrer et al. [28] show how to move from domain-specific language to smart contract code. On the other hand, Capocasale et al. [29] discuss the issue of standardization of smart contracts in a broader context, independent of the field of application. Various formal verification methods are used in efforts to improve smart contracts. Concerning recent work in this area, Nam et al. [30] propose to analyze Solidity smart contracts using Alternating-time Temporal Logic model checking, whereas Almakhour et al. [31] deal with formal verification of composite smart contracts that require other smart contracts to be executed. They employ the finite state machine models to verify Solidity smart contracts. In contrast, Pasqua et al. [32] introduce the method that analyzes Ethereum bytecode and extract precise Control-Flow Graphs.

From the point of view of software engineering, it is important to be able to reuse already written source code. In this regard, Pierro et al. [33] propose to organize a repository of Ethereum smart contracts. Smart contract source code reuse is also discussed by Chen et al. [34]. Their research reveals that about 26% of smart contract source code blocks are reused in 146,452 analyzed open-source Ethereum projects, whereas Khan et al. [35] introduce the topic of source code cloning of Ethereum smart contracts. In this aspect, Górski [11] presents a design pattern that enables the reuse of validation rules used in a smart contract. The usage of classes to define verification rules enables the reuse of rules within the same contract and between different contracts. It should be also emphasized that the self-adaptation characteristic is not sufficiently researched in the context of smart contracts. However, scientific work is emerging in this area. Singh et al. [36] propose a self-adaptive security approach for smart contracts based on Service Level Agreements to provide countermeasures to attacks. Looking even more broadly from the point of view of software architecture, smart contracts are considered an important type of IT system function. The 1 + 5 architectural view model for cooperating systems proposed by Górski in 2012 already included the *Contracts* view [37]. However, it was only at the EUROCAST conference in February 2019 that the same author presented the context of using the *Contracts* view for smart contracts [38,39]. In this model, the key is to obtain business justification for the functions considered in the IT system. The business process is modeled in the *Integrated Processes* view. A similar aspect was underlined at the OTM Conferences in October 2019 by Bagozi et al. [40]. They showed the design of smart contracts identified from business process models of collaborating organizations. In their work, they also used a two-level model of smart contract design: abstract and concrete. This confirms the validity of the adopted architecture for the *AdapT* pattern.

The currently proposed *AdapT* v2.0 design pattern takes source code reuse to an even higher level. Validation rule objects are shared at runtime. Thanks to this, their redundancy was eliminated. In consequence, the efficiency of memory usage has been raised. The pattern is also now adapted to support any number of transaction types. As a result, the pattern has also been made more flexible as far as self-adaptation is concerned.

3. The Pattern Design and Implementation

Further considerations require clarification of the terms used in the paper, i.e., the verification rule, verification rule object list, smart contract configuration, and evaluation expression. The author has proposed the following definition of a verification rule (Definition 1).

Definition 1 (Verification rule). *A single logical condition imposed on a smart contract.*

Smart contracts may employ numerous verification rules. Moreover, the author has introduced the following definition of a smart contract verification rule object list (Definition 2).

Definition 2 (Verification rule object list). *An ordered collection of all non-recurring verification rule objects for all verification rules used in the smart contract.*

Where a smart contract supports multiple transaction types, verification rule configurations apply. Therefore, the author has put forward the following definition of a verification rule configuration (Definition 3).

Definition 3 (Verification rule configuration). *An ordered list of non-repeating verification rule references that point to verification rule objects appropriate to the transaction type.*

The notions of verification rule configuration and configuration will be used interchangeably hereafter. All verification rules that constitute a configuration must be met for the transaction to be executed. Verification rules in the configuration are used by an evaluation expression. The author has formulated the following definition for the notion of the evaluation expression (Definition 4).

Definition 4 (Evaluation expression). *A logical expression containing verification rules and logical operators that return a single boolean value.*

The pattern was constructed in division into two layers: *Abstract* and *Concrete*. The split was used to introduce an abstraction layer, common to all smart contracts designed according to this scheme. The elements of the layer are independent of the implementation of a specific smart contract and are reused in each of them.

3.1. Abstract Layer

The *Abstract* layer consists of two abstract classes: *AbstractTransaction* and *AbstractSmartContract*. Figure 2 presents a UML Class diagram with both classes in the *Abstract* layer of the *AdapT* pattern.

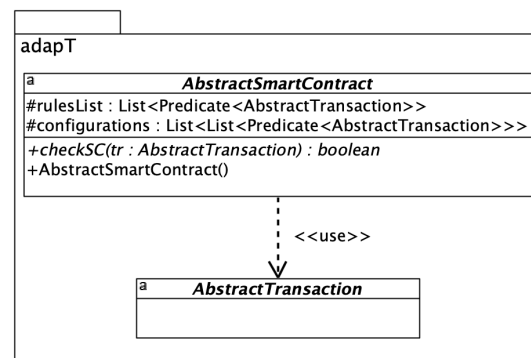


Figure 2. Classes in the *Abstract* layer of the *AdapT* v2.0 pattern.

The *Abstract* layer consists of two abstract classes: *AbstractTransaction* and *AbstractSmartContract*. The *AbstractTransaction* class serves as a parent class for all specific transaction classes handled by the specific smart contract. All specific transaction classes must inherit

from that abstract class, whereas the *AbstractSmartContract* class sets a template for all specific smart contract classes. The class declares a list of verification rule objects (*rulesList* variable). That list employs the *Predicate* functional interface which in turn operates on reference type *AbstractSmartContract*. The class also declares a list of verification rule configurations for various types of transactions (*configurations* variable). Such structure will enable two characteristics: handling various reference types inheriting from *AbstractTransaction*, and the use of lambda expressions for deferred execution. In addition, the *AbstractSmartContract* class declares the *checkSC()* method, which is used to verify the smart contract. It has one input parameter, which is a reference to the transaction object to be verified. This supports the later implementation of *pure* type, where the result hinges only on the input data. Depending on the verification result, this method returns a *true* or *false* logical value. The combination of inheritance from object-oriented programming and lambda expressions from functional programming will allow for processing different types of transactions in this one method. Since they are abstract classes, none of them can be instantiated.

Both classes from the *Abstract* layer of the pattern were implemented in Java language. The source code of the *AbstractSmartContract* class is shown in Listing 1.

Listing 1. The source code of the *AbstractSmartContract* class.

```
package adapt;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public abstract class AbstractSmartContract {
    protected List<Predicate<AbstractTransaction>> rulesList = new ArrayList<>();
    protected List<List<Predicate<AbstractTransaction>>> configurations = new
        ArrayList<>();

    public AbstractSmartContract() {
        configurations.add(new ArrayList<>());
    }
    public boolean checkSC(AbstractTransaction tr){
        boolean correct = false;
        for (Predicate<AbstractTransaction> vR : configurations.get(0)) {
            correct = vR.test(tr);
            if (!correct) break;
        }
        return correct;
    }
}
```

The source code of the *AbstractTransaction* class is shown in Listing 2.

Listing 2. The source code of the *AbstractTransaction* class.

```
package adapt;

public abstract class AbstractTransaction {
}
```

Only standard classes and interfaces available in Java language are used in the implementation. Additionally, the code is written to be domain-independent, whereas the *Concrete* layer of the pattern uses classes implementation-specific for the concrete smart contract.

3.2. Concrete Layer

Smart contracts are widely used in energy applications. Their usages were recently reviewed by Vionis and Kotsilieris [41]. The *Concrete* layer of the pattern uses the example of energy transfer between different stakeholders in the distributed renewable energy system. In such systems, energy can be exchanged between prosumers within the same

community and between prosumers in different communities. Additionally, electricity can be sent to the power grid.

The schema Figure 3 shows a UML Use case diagram for the *SendEnergy* use case. The Use case diagram employs stereotype <<IntegratedSystem>> from the *UML Profile for Messaging Patterns* defined by Górski [42]. The stereotype denotes actors representing applications external to the prosumer one.

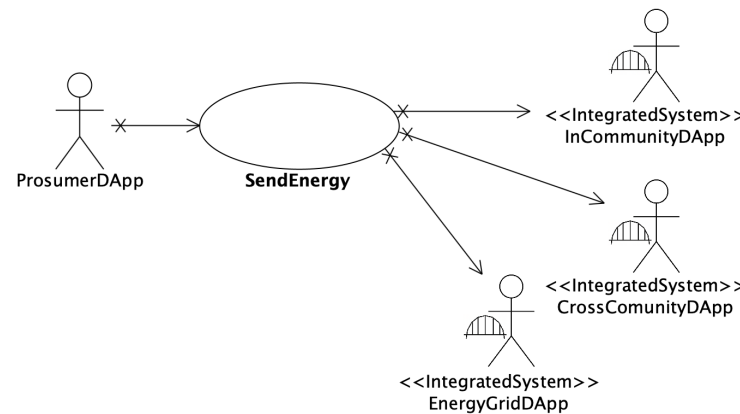


Figure 3. The *SendEnergy* use case with various external users.

The action performed in the use case is the same, but the conditions checked differ depending on the transaction type. The example assumes the following set of verification rules used by the three transaction types considered:

- The source of the transaction must be different from the target of the transaction;
- Energy quantity to transfer must be greater than zero;
- Energy surplus in the source node must be greater or equal energy quantity to transfer;
- Source community must differ from target community;
- Target need must be greater or equal to energy quantity to transfer;
- The target is the subnet energy grid.

Using the *AdapT* pattern, the *Send energy* use case can be implemented with one smart contract. Figure 4 depicts a UML Class diagram with classes that both constitute the abstract layer of the *AdapT* pattern and implement classes of the concrete *ExchangeEnergy* smart contract.

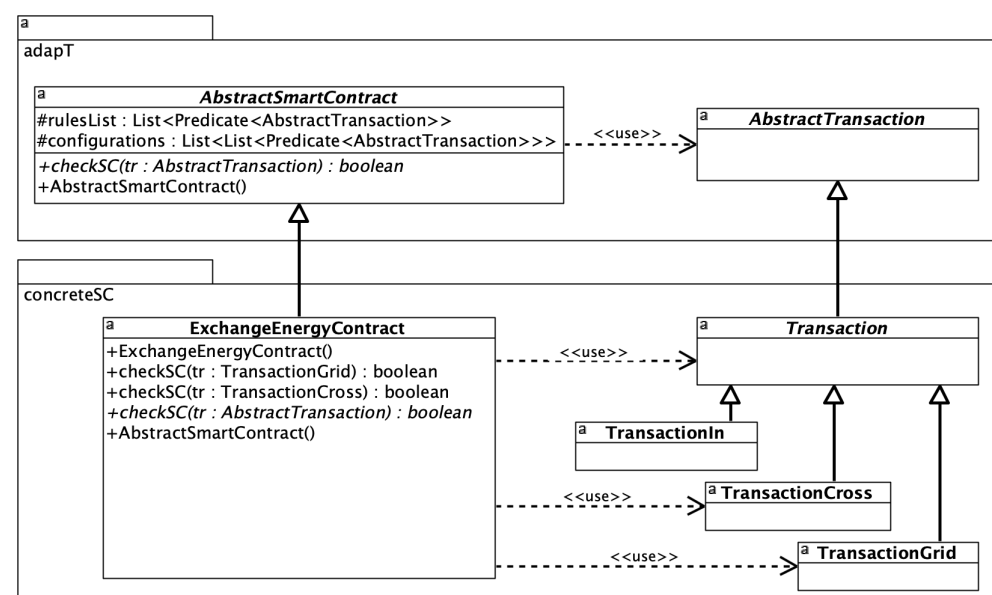


Figure 4. The *AdapT* pattern with abstract and concrete classes.

The concrete smart contract *ExchangeEnergyContract* handles three transaction types: in-community (*TransactionIn* class), cross-community (*TransactionCross* class), and to-grid (*TransactionGrid* class). The drawing Figure 5 presents all classes declared for transaction types in both layers of the pattern for this concrete smart contract.

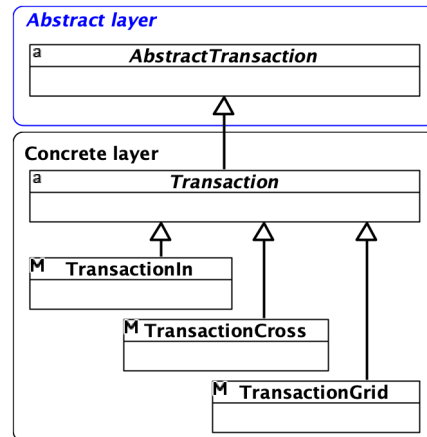


Figure 5. Classes for transaction types for the *SendEnergy* use case.

One additional class the *Transaction* has been defined to gather common attributes of transactions. The class counteracts the redundancy of attributes. The *Transaction* class is also abstract. One can only instantiate objects of the three specific transaction classes. The class that inherits from the *AbstractSmartContract* abstract class is responsible for handling various transaction types. In the presented example the *ExchangeEnergyContract* class inherits from that abstract class. In the constructor of the smart contract class, both the verification rule list and configurations for transaction types are initiated.

The constructor source code of the *ExchangeEnergyContract* class presents Listing 3.

Listing 3. The source code of the *ExchangeEnergyContract* class constructor.

```

public ExchangeEnergyContract() {
    // verification rules
    rulesList.add(t -> ((Transaction) t).getSourceID() != ((Transaction) t).
        getTargetID());
    rulesList.add(t -> ((Transaction) t).getQuantity() > 0);
    rulesList.add(t -> ((Transaction) t).getSourceSurplus() >= ((Transaction) t).
        getQuantity());
    rulesList.add(t -> ((TransactionCross) t).getSourceCommunityID() != ((
        TransactionCross) t).getTargetCommunityID());
    rulesList.add(t -> ((Transaction) t).getTargetNeed() >= ((Transaction) t).
        getQuantity());
    rulesList.add(t -> ((TransactionGrid) t).getTargetID() == ((TransactionGrid) t
        ).getEnergySubnetID());
    // configurations
    for (int i = 1; i <= 2 ; i++) configurations.add(new ArrayList<>());
    // configure rules for TransactionIn
    configurations.get(0).add(rulesList.get(0));
    configurations.get(0).add(rulesList.get(1));
    configurations.get(0).add(rulesList.get(2));
    // configure rules for TransactionGrid
    configurations.get(1).add(rulesList.get(0));
    configurations.get(1).add(rulesList.get(1));
    configurations.get(1).add(rulesList.get(2));
    configurations.get(1).add(rulesList.get(5));
    // configure rules for TransactionCross
    configurations.get(2).add(rulesList.get(3));
    configurations.get(2).add(rulesList.get(0));
    configurations.get(2).add(rulesList.get(1));
    configurations.get(2).add(rulesList.get(2));
    configurations.get(2).add(rulesList.get(4));
}
  
```


In the constructor of the *ExchangeEnergyContract* class, the list of verification rule objects is created in the form of lambda expressions. It is worth noting that verification rules 4 and 6 are dedicated to processing objects of the *TransactionCross* and *TransactionGrid* classes, respectively. The constructor also sets configurations as distinct lists of references to verification rule objects for each transaction type. The *checkSC()* method, implemented in the smart contract abstract class, operates on the first configuration of verification rules. In a specific smart contract class, this method should be overloaded as many times as there are additional transaction types. In the example considered, two more methods had to be written. One method for the cross-community transaction type and one for the to-grid transaction type. Importantly, if the *checkSC()* method is called with a transaction type other than declared for the smart contract, it will execute correctly and return a *false* logical value. The *true* logical value, which proves the correct verification of the transaction, may be returned only for one of the considered transaction types.

The source code of the *checkSC()* method for handling *TransactionGrid* transactions was shown in Listing 4.

Listing 4. The source code of the *checkSC()* method for to-grid transactions.

```
public boolean checkSC(TransactionGrid tr){
    boolean correct = false;
    for (Predicate<AbstractTransaction> vR : configurations.get(1)) {
        correct = vR.test(tr);
        if (!correct) break;
    }
    return correct;
}
```

On invocation of the *checkSC()* method, Java verifies the parameter type and calls that appropriate method from those overloaded.

The calling of the overloaded *checkSC()* method is shown in the UML Sequence diagram (Figure 6).

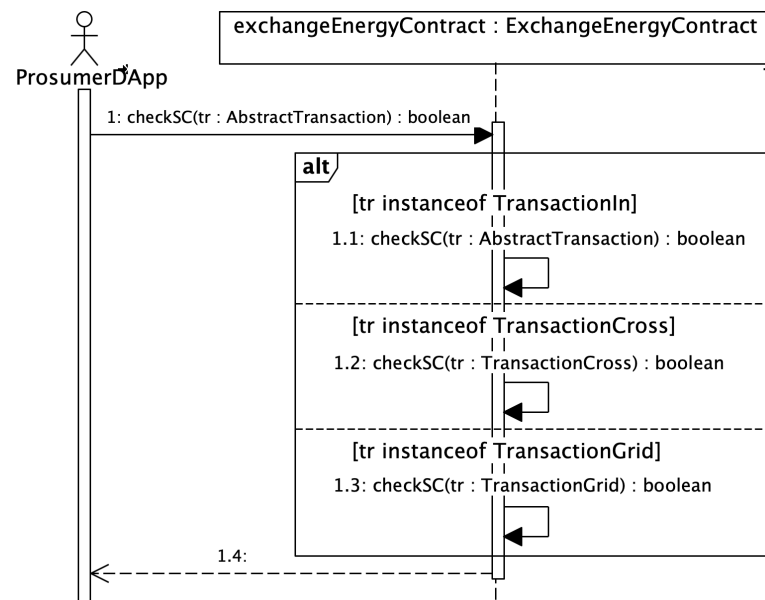


Figure 6. Calling the smart contract verification method.

The order in which the verification rules appear in the configuration matters, because the *checkSC()* method evaluates the verification rules in the order they were set in the configuration. Evaluation of the configuration is aborted if a single verification rule is not met. Such a way of evaluation shortens the smart contract checking time.

The source code of the *Transaction* class is presented in Listing 5.

Listing 5. The source code of the *Transaction* class.

```
package concreteSC;

import adapT.AbstractTransaction;

public abstract sealed class Transaction extends AbstractTransaction permits
    TransactionIn, TransactionCross, TransactionGrid {
    private double quantity;
    private double sourceSurplus;
    private double targetNeed;
    private double targetProduction;
    private double targetBatteryEnergySurplus;
    private int sourceID;
    private int targetID;

    public Transaction(double quantity, double sSurplus, double tNeed, double
        targetProduction, double targetBatteryEnergySurplus, int sID, int tID) {
        this.quantity = quantity;
        this.sourceSurplus = sSurplus;
        this.targetNeed = tNeed;
        this.targetProduction = targetProduction;
        this.targetBatteryEnergySurplus = targetBatteryEnergySurplus;
        this.sourceID = sID;
        this.targetID = tID;
    }
    public double getQuantity() { return quantity; }
    public double getSourceSurplus() {
        return sourceSurplus; }
    public double getTargetNeed() { return targetNeed; }
    public int getSourceID() { return sourceID; }
    public int getTargetID() { return targetID; }
    public double getTargetProduction() {
        return targetProduction; }
    public double getTargetBatteryEnergySurplus() { return
        targetBatteryEnergySurplus; }
}
```

The *Transaction* class is marked as sealed. The possibility of inheriting from the class is permitted only for transaction classes that should be handled by this smart contract. The inheritance tree was closed by marking descendant classes as final ones. Both features are visible in the source code of the *TransactionCross* class shown in Listing 6.

Listing 6. The source code of the *TransactionCross* class.

```
package concreteSC;

public final class TransactionCross extends Transaction {
    private int sourceCommunityID;
    private int targetCommunityID;
    public TransactionCross(double quantity, double sSurplus, double tNeed,
        double targetProduction, double targetBatteryEnergySurplus, int sID, int
        tID, int sCID, int tCID) {
        super(quantity, sSurplus, tNeed, targetProduction,
            targetBatteryEnergySurplus, sID, tID);
        this.sourceCommunityID = sCID;
        this.targetCommunityID = tCID;
    }
    public int getSourceCommunityID() { return sourceCommunityID; }
    public int getTargetCommunityID() { return targetCommunityID; }
}
```

The *TransactionIn* and *TransactionGrid* classes were implemented in the same way as the class for cross-community transaction type. Both classes inherit from the *Transaction* class and terminate inheritance by employing the final keyword. The use of the *Transaction* class, in *rulesList* variable, also allows the processing of a list of rules on each of the specific types of transactions by the same method.

The source code of the *AdapT* v2.0 design pattern is available in the publicly accessible GitHub repository [43].

4. Reuse and Redundancy Analysis

The effectiveness of the use of the software source code affects its maintenance. Raising the level of its reuse facilitates modifications and reduces the scope of testing. The author has introduced a measure U^{sc} as the percentage of reused verification rules under a smart contract. The U^{sc} is expressed by Equation (1).

$$U^{sc} = \frac{\sum_{i=1}^C (\frac{u_i^r}{u_i})}{C} \times 100 \quad (1)$$

where:

C —the number of configurations in the smart contract;

u_i^r —the number of reused verification rules in i -th configuration in the smart contract;

u_i —the number of verification rules in i -th configuration in the smart contract.

The value of the source code efficiency reuse U^{sc} for the smart contract considered in

the article was calculated: $U^{sc} = \frac{(0 + 1 + \frac{3}{4})}{3} \times 100 = \frac{7}{12} \times 100 = 58.3\%$.

Configurations were taken for calculations in order from the most numerous to the least numerous. A score above 50% indicates a high level of source code reuse of validation rules. It should be remembered that if the design pattern was not applied, such would be the level of redundancy of verification rules.

At run-time, it is also worth determining the level of efficiency of using the set of verification rule objects in configurations. The author has proposed the measure D^{sc} as the percentage of redundant verification rule objects for a smart contract. The measure can be expressed by Equation (2).

$$D^{sc} = \frac{(\sum_{i=1}^C o_i^c) - v^{sc}}{v^{sc}} \times 100 \quad (2)$$

where:

o_i^c —number of newly created objects in i -th configuration of the smart contract;

v^{sc} —number of unique verification rules in the smart contract.

The value of the percentage of redundant verification rule objects D^{sc} for the smart contract considered in the article was calculated: $D^{sc} = \frac{(3 + 1 + 2) - 6}{6} \times 100 = 0\%$.

This means that verification rule objects are fully used by configurations. In addition, making changes in checking between transaction types does not create new objects or drop existing ones. As a result, the garbage collector is not involved in the operation of the software. It should also be underlined that no verification rule object from the verification rule object list is left unused.

In a recently published research paper, Khan et al. [35] showed that the overall cloning rate of Solidity smart contracts is 30.13%, of which 27.03% are exact duplicates. The *AdapT* pattern employed to design the smart contract allows for achieving a verification rule cloning rate of 0%.

Therefore, developing design patterns that increase the reuse level of the source code of smart contracts seems to be one of the appropriate directions of research work.

In addition, the efficiency of the run-time use of operational memory is of great importance. Working with various transaction types may involve the constant instantiation of many extra objects. Especially when transactions of various types are evaluated alternately. The author has proposed the measure of object creation efficiency O_T^{sc} as the mean value of the number of verification rule objects created at runtime in the smart contract for a single transaction. The measure can be expressed by Equation (3).

$$O_T^{sc} = \frac{\sum_{i=1}^T o_i^t}{T} \quad (3)$$

where:

T —the number of checked transactions;

o_i^t —the number of newly created objects for i -th transaction.

Calculated values of O_T^{sc} for the considered smart contract for selected quantities of checked transactions were presented in Table 1.

Table 1. Values of O_T^{sc} for the considered smart contract.

No of Transactions	O_T^{sc}
1	6
100	0.06
10,000	0.0006
1,000,000	0.000006
100,000,000	0.00000006

The value of the measure of object creation efficiency O_T^{sc} decreases for the considered smart contract as the number of checked transactions increases. It stems from the fact that the complete set of verification rule objects is created when a smart contract object is instantiated. All transactions are served by the same collection of objects regardless of the number of transactions. For the smart contract considered in the article, six objects are created, one for each of the verification rules. That set of six verification rule objects verifies any number of transactions. No other objects are created for verification rules.

5. Performance Tests Results

The purpose of the performance tests was to check how quickly transactions are checked by a smart contract designed following the *AdapT* pattern. As far as the execution environment is concerned, the tests were carried out on a MacBook Air with the Apple M2 processor, 16 GB Random Access Memory (RAM), and 256 GB Solid State Drive (SSD). The MacBook Air worked under the macOS Sonoma 14.3 operating system. To conduct performance tests, a separate *TestContract* smart contract testing class was designed. The class contains two methods: *conductTest()* and *runTests()*. The first method measures the evaluation time of a smart contract. The method uses the *System.nanoTime()* to obtain an accuracy of one nanosecond of time measurement. The second method is responsible for performing the appropriate number of measurement repetitions. The source code of the *TestContract* class is shown in Listing 7.

The evaluation time of a specific number of transactions by the smart contract E_n^{sc} was adopted as the basic performance measure. Each test has been repeated 50 times to obtain the mean value of E_n^{sc} . Tests were conducted for the smart contracts with 3, 4, and 5 verification rules. A total of 450 tests were run.

Figure 7 depicts test results for the number of transactions in the following range, $n \in \langle 100,000; 10,000,000 \rangle$. The results in the figure are presented on a logarithmic scale.

Listing 7. The source code of the *TestContract* class.

```

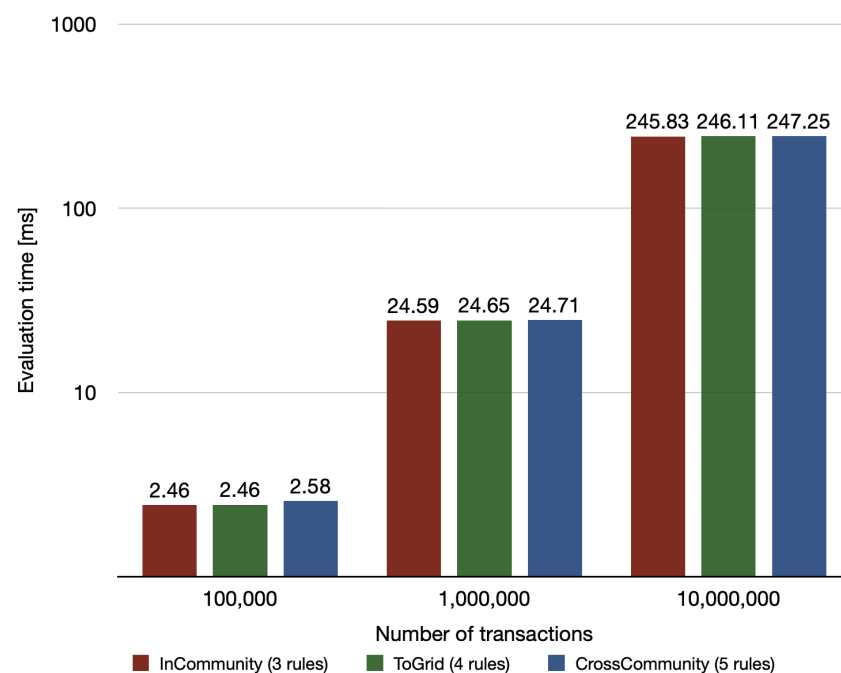
package tests;

import adapT.AbstractSmartContract;
import adapT.AbstractTransaction;

public class TestContract {
    private static int[] numberTransactions = new int[]{100000, 1000000,
        10000000};
    private static long conductTest(AbstractSmartContract sC,
        AbstractTransaction tR, int numberTransactions, int repetitions){
        boolean correct = false;
        long sum = 0;
        long startTime = 0, endTime = 0;
        for (int j = 0; j <= repetitions; j++){
            startTime = System.nanoTime();
            for (int i = 0; i < numberTransactions; i++) correct = sC.checkSC(tR
                );
            endTime = System.nanoTime();
            if( j > 0 ) sum = sum + (endTime - startTime);
        }
        return sum/repetitions;
    }

    public static void runTests(AbstractSmartContract sC, AbstractTransaction tR
        , int repetitions){
        long executionTime = 0;
        // smart contract evaluation time
        System.out.println("Evaluation time of transaction: " + tR.getClass())
            ;
        for (int numberTransaction : numberTransactions) {
            System.out.println("Number of processed transactions: " +
                numberTransaction);
            executionTime = conductTest(sC, tR, numberTransaction, repetitions);
            System.out.println("Execution time: " + executionTime);
        }
    }
}

```

**Figure 7.** Smart contract evaluation time, $n \in \langle 100,000; 10,000,000 \rangle$.

One of the facts that stem from the results presented in Figure 7 deserves to be emphasized. The evaluation time of 10,000,000 transactions is below 0.25 s, regardless of the considered number of verification rules in the smart contract. Currently, the Solana blockchain framework is regarded as one of the quickest offering a throughput of up to several dozen thousand transactions per second [44]. The results obtained illustrate the performance potential of smart contracts designed according to the pattern.

It is also worth visualizing the mean evaluation time of a single transaction. Figure 8 shows the values for this measure calculated from the results obtained when evaluating the considered transaction volumes. Results are shown on a linear scale and are expressed in nanoseconds.

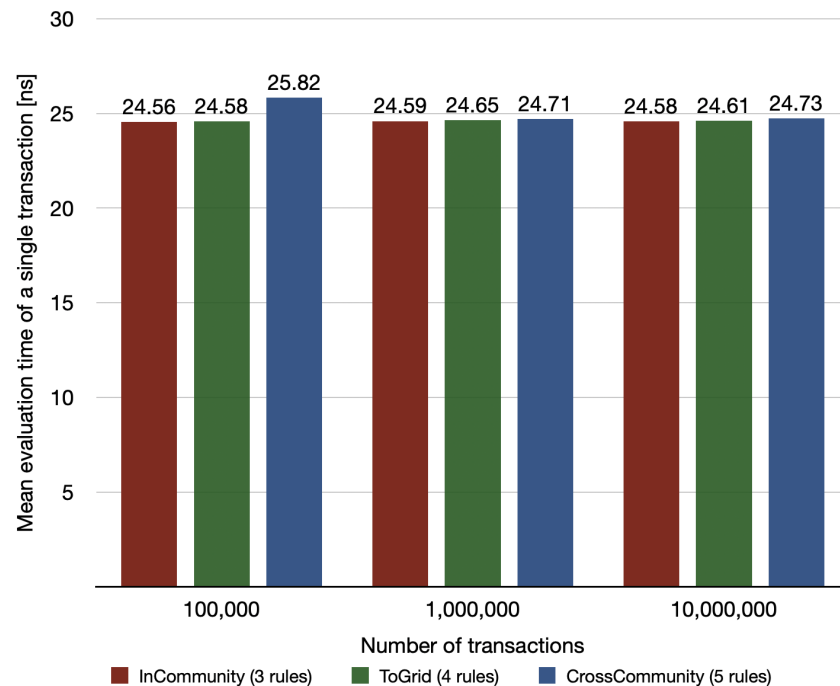


Figure 8. Smart contract evaluation time, $T \in \langle 100,000; 10,000,000 \rangle$.

The average time to check a single transaction is practically constant and is approximately 25 nanoseconds. This means that the smart contract validation mechanism in the pattern has been designed correctly. The increase in evaluation time is directly proportional to the number of transactions evaluated.

6. Discussion and Limitations

In the construction of the pattern, two layers have been distinguished: abstract and specific to a particular smart contract. The use of an abstract layer is a recommended software design practice. Recently, Spray et al. [45] have shown the positive impact of the Abstraction Layered Architecture on software reusability and testability. The abstract layer of the *AdapT* pattern is free of implementation-specific classes or interfaces. The layer consists of two abstract classes and uses only interfaces from Java Standard Edition packages, i.e., *List* and *Predicate*. The *VerificationRule* interface is not needed anymore. Instead, verification rules are handled by the *Predicate* functional interface. Additionally, the abstract smart contract class is independent of any specific transaction class. Generally, both abstract classes in the abstract layer are in this version of the pattern completely independent of the implementation of a specific smart contract. Thanks to this, those abstract classes are reusable without the need for any changes.

The construction of the concrete layer has also been simplified. Currently, no class is needed for any verification rule in the concrete layer. Verification rules are stored as lambda expressions using the *Predicate* interface. It is worth adding that they are stored

in one variable, which makes maintaining verification rules within the smart contract easier. Such a manner of verification rule design makes smart contracts less prone to errors. The construction of the concrete layer reduces the number of classes and raises software maintainability.

The pattern also employs polymorphism as one of the basic object-oriented paradigms. The transaction verification method the *checkSC()* was overloaded for the considered transaction types. The usage of overloaded methods simplified the source code and reduced the number of operations needed. The overloaded methods eliminated conditional statements from the implementation of the verification mechanism invocation.

Java was used because it is a general-purpose language. The development community of this language is large. Additionally, there are available open-source components written in Java. The language itself allows the use of both object-oriented and functional programming structures. Java is also used to program smart contracts on Corda, Hyperledger Fabric, IBM Blockchain, Ethereum, and Neo.

The pattern reduces the redundancy of verification rule objects at run-time to zero. Verification rules are reused among configurations within the single smart contract. The possibility of reusing verification rules between smart contracts was not considered. The paper deals with configurations. In the previous version of the pattern, there were only two possible configurations for two transaction types. Now the design allows for flexibly adding and handling any number of configurations. Configurations may find many applications. Configurations may be applied to handle on-chain and off-chain transactions realized by the same smart contract. Blockchain is also used for authentication and configurations of a smart contract can be used to authenticate various roles. Additionally, the idea of the reuse of verification rules may be adopted for logical conditions in various methods of a smart contract. This should generally increase the maintainability of smart contract source code.

The accuracy of the obtained smart contract execution time is at the level of 1 nanosecond. Because the evaluation time of a single transaction is approximately twenty-five nanoseconds, the measurement precision is sufficient. Especially since the aim was to show the behavior of the smart contract evaluation mechanism with significant transaction volumes.

7. Conclusions

The article presents a smart contract design pattern that allows for handling many types of transactions. The paper contains both the design of the pattern and its implementation in Java language. The pattern's structure combines object-oriented and functional programming mechanisms. The adoption of *sealed* classes increases the security of processed transactions. The *Predicate<T>* functional interface and lambda expressions were employed to reduce the number of classes. As a result, software maintenance costs are also lowered. The design of the pattern allows the verification rules to be reused within the smart contract. The usage of the pattern ensures that the redundancy ratio of verification rule objects at runtime is zero. It also allows for the effective use of operating memory by the smart contract. The implementation of the pattern is independent of the blockchain platform. As a result, it was possible to conduct performance tests of the written software independent of other elements of the blockchain technology, in particular the consensus algorithm. The tests were carried out for a wide range of the number of processed transactions. The evaluation time of a smart contract for <100,000; 10,000,000> transactions takes between <0.0025; 0.25> seconds. Importantly, the analysis of the tests showed that the increase in the evaluation time is directly proportional to the number of transactions checked. The performance test results clearly show that smart contracts as software have great potential for processing large volumes of transactions, far beyond the capabilities of currently available environments.

In the context of further work, Rust language is gaining increasing attention in the community. A general-purpose language may have various applications. Blockchain is one of them. Rust enforces memory safety and a restrictive model of data ownership. The language eliminates the need for a garbage collector. Additionally, Rust supports

a Zero-Cost Abstraction paradigm, which facilitates gaining an efficient compiled code. The language promises a high performance. Thus, the work is going on implementing the pattern in the Rust language and testing smart contracts in the Solana framework. Moreover, the current version of the pattern shares verification rules within one smart contract. It is planned to improve the design pattern to enable sharing verification rules among various smart contracts. It would be also beneficial to move generalized types of verification rules into the abstract layer of the pattern.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

Conflicts of Interest: The author declares no conflicts of interest.

References

1. Zheng, Z.; Xie, S.; Dai, H.-N.; Chen, W.; Chen, X.; Weng, J.; Imran, M. An overview on smart contracts: Challenges, advances and platforms. *Future Gener. Comput. Syst.* **2020**, *105*, 475–491. [CrossRef]
2. Wu, C.; Xiong, J.; Xiong, H.; Zhao, Y.; Yi, W. A Review on Recent Progress of Smart Contract in Blockchain. *IEEE Access* **2022**, *10*, 50839–50863. [CrossRef]
3. Kannengießner, N.; Lins, S.; Sander, C.; Winter, K.; Frey, H.; Sunyaev, A. Challenges and Common Solutions in Smart Contract Development. *IEEE Trans. Softw. Eng.* **2022**, *48*, 4291–4318. [CrossRef]
4. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed.; Addison-Wesley Professional: Upper Saddle River, NJ, USA, 1995.
5. Six, N.; Herbaut, N.; Salinesi, C. Blockchain software patterns for the design of decentralized applications: A systematic literature review. *Blockchain Res. Appl.* **2022**, *3*, 100061. [CrossRef]
6. Liu, J.; Li, P.; Cheng, R.; Asokan, N.; Song, D. Parallel and Asynchronous Smart Contract Execution. *IEEE Trans. Parallel Distrib. Syst.* **2022**, *33*, 1097–1108. [CrossRef]
7. Gupta, A.; Gupta, R.; Jadav, D.; Tanwar, S.; Kumar, N.; Shabaz, M. Proxy smart contracts for zero trust architecture implementation in Decentralised Oracle Networks based applications. *Comput. Commun.* **2023**, *206*, 10–21. [CrossRef]
8. Kim, K.; Ryu, J.; Lee, H.; Lee, Y.; Won, D. Distributed and Federated Authentication Schemes Based on Updatable Smart Contracts. *Electronics* **2023**, *12*, 1217. [CrossRef]
9. Upgrading Smart Contracts in Ethereum. Available online: <https://ethereum.org/en/developers/docs/smart-contracts/upgrading/> (accessed on 7 February 2024).
10. Chu, H.; Zhang, P.; Dong, H.; Xiao, Y.; Ji, S.; Li, W. A survey on smart contract vulnerabilities: Data sources, detection and repair. *Inf. Softw. Technol.* **2023**, *159*, 107221. [CrossRef]
11. Górski, T. Reconfigurable Smart Contracts for Renewable Energy Exchange with Re-Use of Verification Rules. *Appl. Sci.* **2022**, *12*, 5339. [CrossRef]
12. Zou, W.; Lo, D.; Kochhar, P.S.; Le, X.D.; Xia, X.; Feng, Y.; Chen, Z.; Xu, B. Smart Contract Development: Challenges and Opportunities. *IEEE Trans. Softw. Eng.* **2021**, *47*, 2084–2106. [CrossRef]
13. Vacca, A.; Di Sorbo, A.; Visaggio, C.A.; Canfora, G. A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. *J. Syst. Softw.* **2021**, *174*, 110891. [CrossRef]
14. Li, J.; Zhao, Z.; Su, Z.; Meng, W. Gas-expensive patterns detection to optimize smart contracts. *Appl. Soft Comput.* **2023**, *145*, 110542. [CrossRef]
15. Mandarino, V.; Pappalardo, G.; Tramontana, E. Some Blockchain Design Patterns for Overcoming Immutability, Chain-Boundedness, and Gas Fees. In Proceedings of the 3rd Asia Conference on Computers and Communications (ACCC), Shanghai, China, 16–18 December 2022; pp. 65–71. [CrossRef]
16. Liu, X.; Muhammad, K.; Lloret, J.; Chen, Y.-W.; Yuan, S.-M. Elastic and cost-effective data carrier architecture for smart contract in blockchain. *Future Gener. Comput. Syst.* **2019**, *100*, 590–599. [CrossRef]
17. Jin, H.; Wang, Z.; Wen, M.; Dai, W.; Zhu, Y.; Zou, D. Aroc: An Automatic Repair Framework for On-Chain Smart Contracts. *IEEE Trans. Softw. Eng.* **2022**, *48*, 4611–4629. [CrossRef]
18. Gec, S.; Stankovski, V.; Lavbič, D.; Kochovski, P. A Recommender System for Robust Smart Contract Template Classification. *Sensors* **2023**, *23*, 639. [CrossRef]
19. Mao, D.; Wang, F.; Wang, Y.; Hao, Z. Visual and User-Defined Smart Contract Designing System Based on Automatic Coding. *IEEE Access* **2019**, *7*, 73131–73143. [CrossRef]

20. Bodorik, P.; Liu, C.G.; Jutla, D. TABS: Transforming automatically BPMN models into blockchain smart contracts. *Blockchain Res. Appl.* **2023**, *4*, 100115. [\[CrossRef\]](#)
21. Shen, X.; Li, W.; Xu, H.; Wang, X.; Wang, Z. A Reuse-Oriented Visual Smart Contract Code Generator for Efficient Development of Complex Multi-Party Interaction Scenarios. *Appl. Sci.* **2023**, *13*, 8094. [\[CrossRef\]](#)
22. Jurgelaitis, M.; čeponienė, L.; Butkienė, R. Solidity Code Generation From UML State Machines in Model-Driven Smart Contract Development. *IEEE Access* **2022**, *10*, 33465–33481. [\[CrossRef\]](#)
23. Yang, Y.; Shi, R.-H.; Li, K.; Wu, Z.; Wang, S. Multiple access control scheme for EHRs combining edge computing with smart contracts. *Future Gener. Comput. Syst.* **2022**, *129*, 453–463. [\[CrossRef\]](#)
24. Honari, K.; Rouhani, S.; Falak, N.E.; Liu, Y.; Li, Y.; Liang, H.; Dick, S.; Miller, J. Smart Contract Design in Distributed Energy Systems: A Systematic Review. *Energies* **2023**, *16*, 4797. [\[CrossRef\]](#)
25. Wang, H.; Guo, C.; Cheng, S. LoC—A new financial loan management system based on smart contracts. *Future Gener. Comput. Syst.* **2019**, *100*, 648–655. [\[CrossRef\]](#)
26. Saim, M.; Mamoon, M.; Shah, I.; Samad, A. E-Voting via Upgradable Smart Contracts on Blockchain. In Proceedings of the International Conference on Futuristic Technologies (INCOFT), Belgaum, India, 25–27 November 2022; pp. 1–6. [\[CrossRef\]](#)
27. Natanelov, V.; Cao, S.; Foth, M.; Dulleck, U. Blockchain smart contracts for supply chain finance: Mapping the innovation potential in Australia-China beef supply chains. *J. Ind. Inf. Integr.* **2022**, *30*, 100389. [\[CrossRef\]](#)
28. Wohrer, M.; Zdun, U. From Domain-Specific Language to Code: Smart Contracts and the Application of Design Patterns. *IEEE Softw.* **2020**, *37*, 37–42. [\[CrossRef\]](#)
29. Capocasale, V.; Perboli, G. Standardizing Smart Contracts. *IEEE Access* **2022**, *10*, 91203–91212. [\[CrossRef\]](#)
30. Nam, W.; Kil, H. Formal Verification of Blockchain Smart Contracts via ATL Model Checking. *IEEE Access* **2022**, *10*, 8151–8162. [\[CrossRef\]](#)
31. Almakhour, M.; Sliman, L.; Samhat, A.E.; Mellouk, A. A formal verification approach for composite smart contracts security using FSM. *J. King Saud Univ.-Comput. Inf. Sci.* **2023**, *35*, 70–86. [\[CrossRef\]](#)
32. Pasqua, M.; Benini, A.; Contro, F.; Crosara, M.; Dalla Preda, M.; Ceccato, M. Enhancing Ethereum smart-contracts static analysis by computing a precise Control-Flow Graph of Ethereum bytecode. *J. Syst. Softw.* **2023**, *200*, 111653. [\[CrossRef\]](#)
33. Pierro, G.A.; Tonelli, R.; Marchesi, M. An Organized Repository of Ethereum Smart Contracts' Source Codes and Metrics. *Future Internet* **2020**, *12*, 197. [\[CrossRef\]](#)
34. Chen, X.; Liao, P.; Zhang, Y.; Huang, Y.; Zheng, Z. Understanding Code Reuse in Smart Contracts. In Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 9–12 March 2021; pp. 470–479. [\[CrossRef\]](#)
35. Khan, F.; David, I.; Varro, D.; McIntosh, S. Code Cloning in Smart Contracts on the Ethereum Platform: An Extended Replication Study. *IEEE Trans. Softw. Eng.* **2022**, *49*, 2006–2019. [\[CrossRef\]](#)
36. Singh, I.; Lee, S.-W. Self-Adaptive Security for SLA Based Smart Contract. In Proceedings of the IEEE 29th International Requirements Engineering Conference Workshops (REW), Notre Dame, IN, USA, 20–24 September 2021; pp. 388–393. [\[CrossRef\]](#)
37. Górski, T. Architectural View Model for an Integration Platform. *J. Theor. Appl. Comput. Sci.* **2012**, *10*, 25–34.
38. Górski, T. Verification of Architectural Views Model 1+5 Applicability. In *Computer Aided Systems Theory—EUROCAST 2019*; Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A., Eds.; Springer: Cham, Switzerland, 2020; LNCS; Volume 12013, pp. 499–506. [\[CrossRef\]](#)
39. Górski, T.; Bednarski, J. Modeling of Smart Contracts in Blockchain Solution for Renewable Energy Grid. In *Computer Aided Systems Theory—EUROCAST 2019*; Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A., Eds.; Springer: Cham, Switzerland, 2020; LNCS; Volume 12013, pp. 507–514. [\[CrossRef\]](#)
40. Bagozi, A.; Bianchini, D.; De Antonellis, V.; Garda, M.; Melchiori, M. A Three-Layered Approach for Designing Smart Contracts in Collaborative Processes. In *On the Move to Meaningful Internet Systems: OTM 2019 Conferences*; OTM 2019; Lecture Notes in Computer Science; Panetto, H., Debruyne, C., Hepp, M., Lewis, D., Ardagna, C., Meersman, R., Eds.; Springer: Cham, Switzerland, 2019; Volume 11877. [\[CrossRef\]](#)
41. Vionis, P.; Kotsilieris, T. The Potential of Blockchain Technology and Smart Contracts in the Energy Sector: A Review. *Appl. Sci.* **2024**, *14*, 253. [\[CrossRef\]](#)
42. Górski, T. Integration Flows Modeling in the Context of Architectural Views. *IEEE Access* **2023**, *11*, 35220–35231. [\[CrossRef\]](#)
43. AdapT v2.0 Smart Contract Design Pattern, GitHub Repository. Available online: <https://github.com/drGorski/AdapT/releases/tag/v2.0> (accessed on 29 January 2024).
44. Solana Network. Available online: <https://solana.com> (accessed on 7 February 2024).
45. Spray, J.; Sinha, R.; Sen, A.; Cheng, X. Building Maintainable Software Using Abstraction Layering. *IEEE Trans. Softw. Eng.* **2022**, *48*, 4397–4410. [\[CrossRef\]](#)

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.