**Checking for functional correctness, privacy, fairness**

**Prompt:**

Final Requirements for Smart Contract Project

1. Introduction

1.1 Purpose

The purpose of this document is to provide a detailed specification for the development of smart contracts to be used in a blockchain-based system for real estate investment and management. The smart contracts will handle user registration, property listing, transaction processing, document verification, payment handling, and dispute resolution.

1.2 Scope

The smart contracts will automate key processes in real estate transactions, including validations, payments, and record-keeping. They will ensure transparency, security, and efficiency in buying, selling, and leasing properties.

1.3 Context and Benefits

Blockchain technology is chosen for its transparency, security, and efficiency. It ensures all transactions are transparent, reduces fraud risk, and automates processes for faster transactions.

1.4 Examples and Metrics

Goals include achieving 100% transaction transparency, reducing transaction times by 50%, and ensuring data security through advanced encryption.

2. Overall Description

2.1 User Needs

Property Buyers and Sellers:Secure and transparent transaction system.

Real Estate Agents: Platform to manage listings and client interactions.

Legal Entities: Access to immutable records for verification and compliance.

2.2 Assumptions and Dependencies

1. Users have access to a compatible cryptocurrency wallet.
2. Property data is accurately digitized and accessible.

2.3 User Personas

Buyers: Seek reliable property data and secure transactions.

Sellers: Need a platform to list and manage property sales.

Agents: Require tools to manage multiple listings and client communications.

Legal Entities: Need access to immutable transaction records for compliance.

2.4 Dependencies and Contingencies

1. Verification step for inaccurately digitized property data.
2. Guides for setting up and using compatible wallets.

3. Functional Requirements

3.1 Use Case Modeling

3.1.1 Use Case: User Registration and Authentication

Use Case Name: User Registration and Authentication

Actors:

1. Buyer
2. Seller
3. Agent
4. Lawyer

Description:

This use case describes the process by which users register and authenticate themselves to interact with the smart contract system.

Preconditions:

1. The user has internet access and a compatible cryptocurrency wallet.
2. Postconditions:
3. The user is registered and authenticated in the system.

Main Flow:

1. User Initiates Registration: The user accesses the registration portal.
2. Enter Details: The user provides personal details, email, and wallet address.
3. Multi-Factor Authentication: The system sends a verification code to the user's email or phone.
4. Verify Code: The user enters the verification code.
5. Store Credentials: The system securely stores user credentials using encryption.
6. Confirmation: The user receives a registration confirmation.

Alternate Flows:

Invalid Verification Code: The user is prompted to re-enter the correct code.

User Already Registered: The system notifies the user if the email or wallet address is already registered.

3.1.2 Use Case: Property Listing Management

Use Case Name: Property Listing Management

Actors:

1. Seller
2. Agent

Description:

This use case describes how sellers or agents can list properties, set prices, update or remove listings, and manage property search functionalities.

Preconditions:

The user is registered and authenticated.

Postconditions:

The property is listed, updated, or removed as per the user’s actions.

Main Flow:

1. Access Property Listing: The seller/agent accesses the property listing section.
2. Enter Property Details: The user enters details such as property description, price, and terms.
3. Submit Listing: The user submits the listing for approval.
4. Listing Approved: The system verifies and approves the listing.
5. Update/Remove Listing: The user can update or remove the listing as needed.
6. Search Properties: Buyers can search for properties using filters like location and price range.

Alternate Flows:

Incomplete Details: The system prompts the user to complete all required fields.

Approval Pending: The listing is pending until approved by the system.

3.1.3 Use Case: Transaction Handling

Use Case Name:Transaction Handling

Actors:

1. Buyer
2. Seller

Description:

This use case describes the process of automating transactions based on predefined conditions, including the use of escrow services.

Preconditions:

a. The buyer and seller are registered and authenticated.

Postconditions:

1. The transaction is completed when predefined conditions are met.

Main Flow:

1. Initiate Transaction: The buyer initiates a transaction by making an offer.
2. Escrow Service: Funds are placed in escrow until conditions are met.
3. Condition Verification: The system verifies predefined conditions, such as payment and document verification.
4. Release Funds: Funds are released from escrow to the seller upon verification.
5. Generate Receipts: The system generates and stores transaction receipts.

Alternate Flows:

Condition Not Met: The system retains funds in escrow until conditions are met.

Dispute Raised: The system initiates the dispute resolution process.

3.1.4 Use Case: Document Handling and Verification

Use Case Name: Document Handling and Verification

Actors:

1. Buyer

2. Seller

3. Agent

4. Lawyer

Description:

This use case describes the process of uploading, verifying, and accessing legal documents.

Preconditions:

- The user is registered and authenticated.

Postconditions:

- Documents are uploaded, verified, and stored securely.

Main Flow:

1. Upload Document: The user uploads necessary legal documents.
2. Verify Document: Authorized personnel verify the documents.
3. Access Control: The system ensures only authorized personnel can access sensitive documents.
4. Digital Signatures: Digital signatures are used to ensure document authenticity.
5. Store Document: The system securely stores verified documents.

Alternate Flows:

Verification Failed: The document is rejected, and the user is notified.

Unauthorized Access: The system denies access and alerts the user.

3.1.5 Use Case: Payment Processing

Use Case Name: Payment Processing

Actors:

1. Buyer
2. Seller

Description:

This use case describes how cryptocurrency payments are handled, including exchange rate adjustments and receipt generation.

Preconditions:

- The buyer and seller are registered and authenticated.

Postconditions:

- Payments are processed, and transaction history is generated.

Main Flow:

1. Initiate Payment: The buyer initiates a cryptocurrency payment.
2. Exchange Rate Adjustment: The system adjusts payment according to current exchange rates.
3. Transfer Funds: Funds are transferred to the seller.
4. Generate Receipt: The system generates and stores payment receipts.

Alternate Flows:

Payment Failure: The system notifies the buyer and retries the payment.

Dispute Raised: The system initiates the dispute resolution process.

3.1.6 Use Case: Dispute Resolution

Use Case Name: Dispute Resolution

Actors:

1. Buyer
2. Seller
3. Arbitrator

Description:

This use case describes the process of handling disputes through predefined rules or third-party arbitration.

Preconditions:

- A dispute has been raised by a buyer or seller.

Postconditions:

The dispute is resolved according to the predefined rules or arbitration outcome.

Main Flow:

1. Raise Dispute: The user raises a dispute.
2. Initiate Arbitration: The system initiates the arbitration process.
3. Review Dispute: The arbitrator reviews the dispute details.
4. Decision Making: The arbitrator makes a decision based on predefined criteria.
5. Resolve Dispute: The system enforces the decision and resolves the dispute.

Alternate Flows:

Arbitration Rejected: The system notifies the user and suggests alternative dispute resolution methods.

4. Non-functional Requirements

4.1 Security

1. Secure against vulnerabilities (e.g., re-entrancy, overflow/underflow).
2. Regular security audits and updates.
3. Specify quarterly security audits and vulnerability assessments.

4.2 Performance

1. Handle transactions quickly, even during high network congestion.
2. Optimized for minimal gas usage.
3. Process up to 1,000 transactions per minute.
4. Ensure transaction processing time under 2 seconds under normal conditions.

4.3 Scalability

1. Handle growing transaction volumes.
2. Support additional property types and transaction structures.
3. Handle a 10% increase in transaction volume per month without performance degradation.

4.4 Usability

1. User-friendly interface.
2. Comprehensive logs and alerts.
3. Aim for 90% user satisfaction.
4. Retain logs for at least one year.

4.5 External Interface Requirements

User Interfaces:

1. Web interface accessible via browsers, optimized for desktop and mobile.
2. Integration with blockchain wallets (e.g., MetaMask).
3. Design Principles: Consistency, simplicity, responsiveness, and accessibility (WCAG 2.1).

Hardware Interfaces:

- Standard computing device with internet access.

Software Interfaces:

1. Blockchain platform (e.g., Ethereum).
2. IPFS for decentralized file storage.

Communications Interfaces:

* 1. HTTPS for secure communication.
  2. Integration with real estate platforms for data exchange.

4.6 Other Requirements

Regulatory Compliance:

1. Comply with local and international real estate laws and regulations.
2. Ensure adherence to data privacy laws.
3. Implement compliance monitoring and reporting.
4. Use encryption and strict access controls for data privacy.

By following this detailed requirements specification, the smart contract system for real estate investment and management will be robust, secure, and efficient, meeting the needs of all stakeholders involved.

The above provided one is the final requirement for smart contracts check whether it is functionally correct, check for fairness and privacy protection

I need response like

If provided requirement is functionally correct , fair and it has privacy protection highlight in the provided requirement which part or factor address that

else provide me content where should I need to work on in the provided requirement to improve this

functionally correctness, fairness, privacy protection.

**Response:**

**Evaluation of the Final Requirements for Smart Contracts**

Based on the provided requirements specification for the smart contracts intended for real estate investment and management, I have evaluated the document for functional correctness, fairness, and privacy protection. Below is the assessment:

**Functional Correctness**

**Assessment**: The provided requirements appear to be functionally correct, with clear and detailed use cases that cover the core functionalities of the smart contract system. The system is designed to handle user registration, property listing management, transaction handling, document verification, payment processing, and dispute resolution.

**Highlighted Parts**:

* **Use Case Modeling**: The requirements comprehensively cover the key processes such as **User Registration and Authentication**, **Property Listing Management**, **Transaction Handling**, **Document Handling and Verification**, **Payment Processing**, and **Dispute Resolution**. Each use case is well-defined with preconditions, postconditions, main flows, and alternate flows, ensuring the system can handle various scenarios functionally.

**Fairness**

**Assessment**: The requirements demonstrate fairness by ensuring that all users (buyers, sellers, agents, and legal entities) have equitable access to the system's functionalities. Additionally, the dispute resolution process is designed to be impartial, with the use of third-party arbitrators and predefined rules.

**Highlighted Parts**:

* **Dispute Resolution Use Case**: The fairness is explicitly addressed through the inclusion of an **Arbitrator** role and a well-defined process for dispute handling, ensuring that all parties are treated fairly in case of a conflict.
* **User Needs Section**: By identifying the distinct needs of **Property Buyers and Sellers**, **Real Estate Agents**, and **Legal Entities**, the requirements ensure that the platform caters to the specific needs of each user type, promoting fairness.

**Privacy Protection**

**Assessment**: The requirements emphasize privacy protection by incorporating security measures such as encryption, access controls, and secure communication protocols. Additionally, the system is designed to comply with data privacy laws, which is crucial for protecting user information.

**Highlighted Parts**:

* **Security Requirements (Section 4.1)**: The emphasis on **secure against vulnerabilities** and **regular security audits and updates** directly addresses the need for privacy protection.
* **Document Handling and Verification Use Case**: The **Access Control** flow ensures that sensitive documents are accessible only to authorized personnel, protecting user privacy.
* **Regulatory Compliance (Section 4.6)**: The requirement to **comply with local and international real estate laws and regulations** and **ensure adherence to data privacy laws** is a strong indicator of privacy protection being a priority.

**Conclusion**

The provided requirements specification for the smart contract system is functionally correct, fair, and includes strong privacy protection measures. The specific sections highlighted demonstrate that the system is well-designed to meet its intended purpose while ensuring fairness and privacy for all users.

No further improvements are needed regarding functional correctness, fairness, or privacy protection based on the information provided. However, it is essential to continue monitoring and updating the system to address any emerging security threats, regulatory changes, or user needs.

Prompt 2:

Based on your response provided requirement provided is functionally correct, Fair and also address the privacy protection. now give me the updated requirement where the system should address the security threats, regulatory changes and user needs.

Response:

**Updated Requirements for Smart Contract Project**

**1. Introduction**

1.1 Purpose  
The purpose of this document is to provide a detailed specification for the development of smart contracts to be used in a blockchain-based system for real estate investment and management. The smart contracts will handle user registration, property listing, transaction processing, document verification, payment handling, and dispute resolution while ensuring security, regulatory compliance, and adaptability to evolving user needs.

1.2 Scope  
The smart contracts will automate key processes in real estate transactions, including validations, payments, and record-keeping. They will ensure transparency, security, and efficiency in buying, selling, and leasing properties, while remaining flexible enough to adapt to future security threats, regulatory changes, and user needs.

1.3 Context and Benefits  
Blockchain technology is chosen for its transparency, security, and efficiency. It ensures all transactions are transparent, reduces fraud risk, and automates processes for faster transactions. The system will also be designed to address emerging security threats, comply with changing regulations, and adapt to new user needs.

1.4 Examples and Metrics  
Goals include achieving 100% transaction transparency, reducing transaction times by 50%, ensuring data security through advanced encryption, and maintaining compliance with the latest regulations.

**2. Overall Description**

2.1 User Needs

* **Property Buyers and Sellers**: Secure, transparent, and adaptable transaction system.
* **Real Estate Agents**: A platform that not only manages listings and client interactions but also evolves with their growing needs.
* **Legal Entities**: Access to immutable and compliant records for verification, compliance, and audit purposes.

2.2 Assumptions and Dependencies

* Users have access to a compatible cryptocurrency wallet.
* Property data is accurately digitized and accessible.
* The system is continuously updated to address new security threats and regulatory requirements.

2.3 User Personas

* **Buyers**: Seek reliable property data, secure transactions, and assurances that the system adapts to their growing needs.
* **Sellers**: Need a platform to list, manage property sales, and ensure their transactions remain secure and compliant with any new regulations.
* **Agents**: Require tools to manage multiple listings and client communications, with the flexibility to adapt to their changing business environment.
* **Legal Entities**: Need access to immutable and compliant transaction records for audit and regulatory compliance.

2.4 Dependencies and Contingencies

* Verification step for inaccurately digitized property data.
* Guides for setting up and using compatible wallets.
* Continuous monitoring and updating to address security threats and regulatory changes.

**3. Functional Requirements**

3.1 Use Case Modeling  
**3.1.1 Use Case: User Registration and Authentication**

* **Actors**: Buyer, Seller, Agent, Lawyer
* **Description**: This use case describes the process by which users register and authenticate themselves to interact with the smart contract system.
* **Preconditions**: The user has internet access and a compatible cryptocurrency wallet.
* **Postconditions**: The user is registered and authenticated in the system.
* **Main Flow**:
  1. User Initiates Registration: The user accesses the registration portal.
  2. Enter Details: The user provides personal details, email, and wallet address.
  3. Multi-Factor Authentication: The system sends a verification code to the user's email or phone.
  4. Verify Code: The user enters the verification code.
  5. Store Credentials: The system securely stores user credentials using encryption.
  6. Confirmation: The user receives a registration confirmation.
* **Alternate Flows**:
  1. Invalid Verification Code: The user is prompted to re-enter the correct code.
  2. User Already Registered: The system notifies the user if the email or wallet address is already registered.

**3.1.2 Use Case: Property Listing Management**

* **Actors**: Seller, Agent
* **Description**: This use case describes how sellers or agents can list properties, set prices, update or remove listings, and manage property search functionalities.
* **Preconditions**: The user is registered and authenticated.
* **Postconditions**: The property is listed, updated, or removed as per the user’s actions.
* **Main Flow**:
  1. Access Property Listing: The seller/agent accesses the property listing section.
  2. Enter Property Details: The user enters details such as property description, price, and terms.
  3. Submit Listing: The user submits the listing for approval.
  4. Listing Approved: The system verifies and approves the listing.
  5. Update/Remove Listing: The user can update or remove the listing as needed.
  6. Search Properties: Buyers can search for properties using filters like location and price range.
* **Alternate Flows**:
  1. Incomplete Details: The system prompts the user to complete all required fields.
  2. Approval Pending: The listing is pending until approved by the system.

**3.1.3 Use Case: Transaction Handling**

* **Actors**: Buyer, Seller
* **Description**: This use case describes the process of automating transactions based on predefined conditions, including the use of escrow services.
* **Preconditions**: The buyer and seller are registered and authenticated.
* **Postconditions**: The transaction is completed when predefined conditions are met.
* **Main Flow**:
  1. Initiate Transaction: The buyer initiates a transaction by making an offer.
  2. Escrow Service: Funds are placed in escrow until conditions are met.
  3. Condition Verification: The system verifies predefined conditions, such as payment and document verification.
  4. Release Funds: Funds are released from escrow to the seller upon verification.
  5. Generate Receipts: The system generates and stores transaction receipts.
* **Alternate Flows**:
  1. Condition Not Met: The system retains funds in escrow until conditions are met.
  2. Dispute Raised: The system initiates the dispute resolution process.

**3.1.4 Use Case: Document Handling and Verification**

* **Actors**: Buyer, Seller, Agent, Lawyer
* **Description**: This use case describes the process of uploading, verifying, and accessing legal documents.
* **Preconditions**: The user is registered and authenticated.
* **Postconditions**: Documents are uploaded, verified, and stored securely.
* **Main Flow**:
  1. Upload Document: The user uploads necessary legal documents.
  2. Verify Document: Authorized personnel verify the documents.
  3. Access Control: The system ensures only authorized personnel can access sensitive documents.
  4. Digital Signatures: Digital signatures are used to ensure document authenticity.
  5. Store Document: The system securely stores verified documents.
* **Alternate Flows**:
  1. Verification Failed: The document is rejected, and the user is notified.
  2. Unauthorized Access: The system denies access and alerts the user.

**3.1.5 Use Case: Payment Processing**

* **Actors**: Buyer, Seller
* **Description**: This use case describes how cryptocurrency payments are handled, including exchange rate adjustments and receipt generation.
* **Preconditions**: The buyer and seller are registered and authenticated.
* **Postconditions**: Payments are processed, and transaction history is generated.
* **Main Flow**:
  1. Initiate Payment: The buyer initiates a cryptocurrency payment.
  2. Exchange Rate Adjustment: The system adjusts payment according to current exchange rates.
  3. Transfer Funds: Funds are transferred to the seller.
  4. Generate Receipt: The system generates and stores payment receipts.
* **Alternate Flows**:
  1. Payment Failure: The system notifies the buyer and retries the payment.
  2. Dispute Raised: The system initiates the dispute resolution process.

**3.1.6 Use Case: Dispute Resolution**

* **Actors**: Buyer, Seller, Arbitrator
* **Description**: This use case describes the process of handling disputes through predefined rules or third-party arbitration.
* **Preconditions**: A dispute has been raised by a buyer or seller.
* **Postconditions**: The dispute is resolved according to the predefined rules or arbitration outcome.
* **Main Flow**:
  1. Raise Dispute: The user raises a dispute.
  2. Initiate Arbitration: The system initiates the arbitration process.
  3. Review Dispute: The arbitrator reviews the dispute details.
  4. Decision Making: The arbitrator makes a decision based on predefined criteria.
  5. Resolve Dispute: The system enforces the decision and resolves the dispute.
* **Alternate Flows**:
  1. Arbitration Rejected: The system notifies the user and suggests alternative dispute resolution methods.

**4. Non-functional Requirements**

**4.1 Security**

* Secure against vulnerabilities (e.g., re-entrancy, overflow/underflow).
* Regular security audits and updates.
* Specify quarterly security audits and vulnerability assessments.
* **Continuous Threat Monitoring**: Implement real-time threat detection and automated responses to mitigate security risks.
* **Incident Response Plan**: Develop and maintain an incident response plan to quickly address security breaches.

**4.2 Performance**

* Handle transactions quickly, even during high network congestion.
* Optimized for minimal gas usage.
* Process up to 1,000 transactions per minute.
* Ensure transaction processing time under 2 seconds under normal conditions.
* **Performance Monitoring**: Implement continuous performance monitoring to ensure system efficiency and detect potential bottlenecks.

**4.3 Scalability**

* Handle growing transaction volumes.
* Support additional property types and transaction structures.
* Handle a 10% increase in transaction volume per month without performance degradation.
* **Adaptive Scalability**: Implement dynamic scaling mechanisms to automatically adjust to increased loads and maintain performance.

**4.4 Usability**

* User-friendly interface.
* Comprehensive logs and alerts.
* Aim for 90% user satisfaction.
* Retain logs for at least one year.
* **User Feedback Integration**: Regularly collect and analyze user feedback to inform system updates and enhancements.

**4.5 External Interface Requirements**

* **User Interfaces**:
  + Web interface accessible via browsers, optimized for desktop and mobile.
  + Integration with blockchain wallets (e.g., MetaMask).
  + **Design Principles**: Consistency, simplicity, responsiveness, and accessibility (WCAG 2.1).
  + **Regulatory Updates**: Ensure the interface reflects any changes in regulatory requirements, providing users with up-to-date compliance information.
* **Hardware Interfaces**:
  + Standard computing device with internet access.
* **Software Interfaces**:
  + Blockchain platform (e.g., Ethereum).
  + IPFS for decentralized file storage.
  + **Compliance Monitoring**: Integrate tools for monitoring compliance with relevant regulations.
* **Communications Interfaces**:
  + HTTPS for secure communication.
  + Integration with real estate platforms for data exchange.
  + **Regulatory Compliance Alerts**: Notify users of any regulatory updates or changes that may impact their transactions.

**4.6 Other Requirements**

* **Regulatory Compliance**:
  + Comply with local and international real estate laws and regulations.
  + Ensure adherence to data privacy laws.
  + Implement compliance monitoring and reporting.
  + Use encryption and strict access controls for data privacy.
  + **Regulatory Updates**: Continuously monitor and integrate regulatory changes into the system to ensure ongoing compliance.
  + **Compliance Audit Trail**: Maintain a detailed audit trail to document compliance activities and decisions.
* **User Needs Adaptation**:
  + Regularly assess and adapt to changing user needs through continuous system updates.
  + Provide training resources and guides to help users adapt to new features or regulatory requirement.

**Conclusion**

The updated requirements specification now includes additional measures to address emerging security threats, adapt to regulatory changes, and respond to evolving user needs. These enhancements ensure that the smart contract system remains robust, compliant, and user-centric, while maintaining its core functionalities of transparency, security, and efficiency.

**Prompt:** Provided updated requirement from previous response followed by it I have given this

Above provided content is my updated requirement generate UML use case diagram along with that give me class diagram, object diagram and sequence diagram.

Response:

Use case diagram:

Code:

from graphviz import Digraph

# Create the use case diagram

dot = Digraph(comment='Real Estate Blockchain System Use Case Diagram')

# Define the actors

dot.node('Buyer', 'Buyer', shape='ellipse')

dot.node('Seller', 'Seller', shape='ellipse')

dot.node('Agent', 'Agent', shape='ellipse')

dot.node('Lawyer', 'Lawyer', shape='ellipse')

dot.node('Arbitrator', 'Arbitrator', shape='ellipse')

# Define the use cases

dot.node('UserRegistration', 'User Registration and Authentication', shape='ellipse')

dot.node('PropertyListing', 'Property Listing Management', shape='ellipse')

dot.node('TransactionHandling', 'Transaction Handling', shape='ellipse')

dot.node('DocumentHandling', 'Document Handling and Verification', shape='ellipse')

dot.node('PaymentProcessing', 'Payment Processing', shape='ellipse')

dot.node('DisputeResolution', 'Dispute Resolution', shape='ellipse')

# Define the relationships

dot.edge('Buyer', 'UserRegistration')

dot.edge('Seller', 'UserRegistration')

dot.edge('Agent', 'UserRegistration')

dot.edge('Lawyer', 'UserRegistration')

dot.edge('Seller', 'PropertyListing')

dot.edge('Agent', 'PropertyListing')

dot.edge('Buyer', 'TransactionHandling')

dot.edge('Seller', 'TransactionHandling')

dot.edge('Buyer', 'DocumentHandling')

dot.edge('Seller', 'DocumentHandling')

dot.edge('Agent', 'DocumentHandling')

dot.edge('Lawyer', 'DocumentHandling')

dot.edge('Buyer', 'PaymentProcessing')

dot.edge('Seller', 'PaymentProcessing')

dot.edge('Buyer', 'DisputeResolution')

dot.edge('Seller', 'DisputeResolution')

dot.edge('Arbitrator', 'DisputeResolution')

# Render the diagram

dot.render('RealEstateSystem\_use\_case\_diagram', format='png', cleanup=False)

# Display the diagram

from PIL import Image

img = Image.open('RealEstateSystem\_use\_case\_diagram.png')

img.show()

Class Diagram:

Code:

from graphviz import Digraph

# Create the class diagram

dot = Digraph(comment='Real Estate Blockchain System Class Diagram')

# Define the classes

dot.node('UserContract', '''UserContract | {

- users: mapping(address => User)

+ registerUser(address, string)

+ authenticateUser(address) view

+ isUserRegistered(address) view

}''', shape='record')

dot.node('PropertyContract', '''PropertyContract | {

- properties: mapping(uint => Property)

+ addProperty(uint, string, uint)

+ updateProperty(uint, string, uint)

+ removeProperty(uint)

+ getProperty(uint) view

}''', shape='record')

dot.node('TransactionContract', '''TransactionContract | {

- transactions: mapping(uint => Transaction)

+ initiateTransaction(uint, uint, address, address, uint)

+ verifyTransaction(uint)

+ completeTransaction(uint)

}''', shape='record')

dot.node('DocumentContract', '''DocumentContract | {

- documents: mapping(uint => Document)

+ uploadDocument(uint, string)

+ verifyDocument(uint)

+ getDocument(uint) view

}''', shape='record')

dot.node('PaymentContract', '''PaymentContract | {

- payments: mapping(uint => Payment)

+ initiatePayment(uint, address, address, uint)

+ verifyPayment(uint)

+ completePayment(uint)

}''', shape='record')

dot.node('DisputeContract', '''DisputeContract | {

- disputes: mapping(uint => Dispute)

+ raiseDispute(uint, uint, address, address, string)

+ resolveDispute(uint, string)

}''', shape='record')

# Define the relationships

dot.edge('TransactionContract', 'UserContract', arrowhead='none', label='uses')

dot.edge('TransactionContract', 'PropertyContract', arrowhead='none', label='uses')

dot.edge('TransactionContract', 'DocumentContract', arrowhead='none', label='uses')

dot.edge('TransactionContract', 'PaymentContract', arrowhead='none', label='uses')

dot.edge('DisputeContract', 'TransactionContract', arrowhead='none', label='uses')

# Render the diagram

dot.render('RealEstateSystem\_class\_diagram', format='png', cleanup=False)

# Display the diagram

img = Image.open('RealEstateSystem\_class\_diagram.png')

img.show()

Object Diagram:

Code:

from graphviz import Digraph

# Create the object diagram

dot = Digraph(comment='Real Estate Blockchain System Object Diagram')

# Define the objects

dot.node('buyer1', 'Buyer: UserContract', shape='record')

dot.node('seller1', 'Seller: UserContract', shape='record')

dot.node('property1', 'Property1: PropertyContract', shape='record')

dot.node('transaction1', 'Transaction1: TransactionContract', shape='record')

dot.node('document1', 'Document1: DocumentContract', shape='record')

dot.node('payment1', 'Payment1: PaymentContract', shape='record')

dot.node('dispute1', 'Dispute1: DisputeContract', shape='record')

# Define the relationships

dot.edge('buyer1', 'transaction1', label='initiates')

dot.edge('seller1', 'transaction1', label='participates')

dot.edge('property1', 'transaction1', label='is related to')

dot.edge('document1', 'transaction1', label='verifies')

dot.edge('payment1', 'transaction1', label='handles')

dot.edge('dispute1', 'transaction1', label='resolves')

# Render the diagram

dot.render('RealEstateSystem\_object\_diagram', format='png', cleanup=False)

# Display the diagram

img = Image.open('RealEstateSystem\_object\_diagram.png')

img.show()

Sequence Diagram:

Code:

from graphviz import Digraph

# Create the sequence diagram

dot = Digraph(comment='Real Estate Blockchain System Sequence Diagram')

# Define the lifelines (objects)

dot.node('buyer', 'Buyer', shape='record')

dot.node('seller', 'Seller', shape='record')

dot.node('transaction', 'TransactionContract', shape='record')

dot.node('payment', 'PaymentContract', shape='record')

dot.node('dispute', 'DisputeContract', shape='record')

# Define the messages (interactions)

dot.edge('buyer', 'transaction', label='Initiate Transaction')

dot.edge('transaction', 'payment', label='Handle Payment')

dot.edge('payment', 'buyer', label='Confirm Payment')

dot.edge('buyer', 'dispute', label='Raise Dispute', style='dashed')

dot.edge('dispute', 'transaction', label='Resolve Dispute', style='dashed')

dot.edge('transaction', 'seller', label='Complete Transaction')

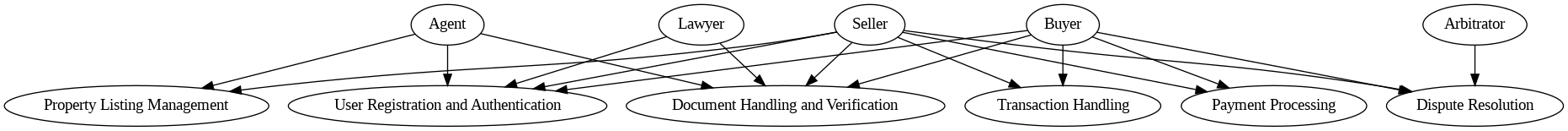
# Render the diagram

dot.render('RealEstateSystem\_sequence\_diagram', format='png', cleanup=False)

# Display the diagram

img = Image.open('RealEstateSystem\_sequence\_diagram.png')

img.show()

Use case diagram:

Class Diagram:

A diagram of a contract

Description automatically generated

Object Diagram:

A diagram of a graph

Description automatically generated

Sequence Diagram:

A diagram of a transaction

Description automatically generated

Implementation:

1. User Registration and Authentication:

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract UserContract {

struct User {

address walletAddress;

string email;

bool isRegistered;

}

mapping(address => User) public users;

event UserRegistered(address indexed user, string email);

function registerUser(string memory email) public {

require(!users[msg.sender].isRegistered, "User already registered");

users[msg.sender] = User({

walletAddress: msg.sender,

email: email,

isRegistered: true

});

emit UserRegistered(msg.sender, email);

}

function isUserRegistered(address userAddress) public view returns (bool) {

return users[userAddress].isRegistered;

}

}

Output:

|  |  |
| --- | --- |
| **status** | 0x1 Transaction mined and execution succeed |
| **transaction hash** | 0x46494edfbb3430b099c96e2c9e29de9aa043930b7a6a468228e02db9430aadc1 |
| **block hash** | 0xd31fcdc28766e63d80f92a02acdc63407c80fc385be05a52a6a9b9caf1288da7 |
| **block number** | 38 |
| **contract address** | 0xB57ee0797C3fc0205714a577c02F7205bB89dF30 |
| **from** | 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 |
| **to** | UserContract.(constructor) |
| **gas** | 711197 gas |
| **transaction cost** | 618432 gas |
| **execution cost** | 524354 gas |
| **input** | 0x608...a0033 |
| **decoded input** | {} |
| **decoded output** | - |
| **logs** | [] |
| **raw logs** | [] |

2. Property Listing Management:

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "./UserContract.sol";

contract PropertyContract {

struct Property {

uint256 id;

address owner;

string description;

uint256 price;

bool isListed;

}

UserContract userContract;

uint256 public propertyCount;

mapping(uint256 => Property) public properties;

mapping(address => uint256[]) public ownerProperties;

event PropertyListed(uint256 propertyId, address indexed owner, string description, uint256 price);

event PropertyUpdated(uint256 propertyId, string description, uint256 price);

event PropertyRemoved(uint256 propertyId);

constructor(address userContractAddress) {

userContract = UserContract(userContractAddress);

}

function listProperty(string memory description, uint256 price) public {

require(userContract.isUserRegistered(msg.sender), "User must be registered");

propertyCount++;

properties[propertyCount] = Property({

id: propertyCount,

owner: msg.sender,

description: description,

price: price,

isListed: true

});

ownerProperties[msg.sender].push(propertyCount);

emit PropertyListed(propertyCount, msg.sender, description, price);

}

function updateProperty(uint256 propertyId, string memory description, uint256 price) public {

Property storage property = properties[propertyId];

require(property.owner == msg.sender, "Only the owner can update this property");

require(property.isListed, "Property is not listed");

property.description = description;

property.price = price;

emit PropertyUpdated(propertyId, description, price);

}

function removeProperty(uint256 propertyId) public {

Property storage property = properties[propertyId];

require(property.owner == msg.sender, "Only the owner can remove this property");

require(property.isListed, "Property is not listed");

property.isListed = false;

emit PropertyRemoved(propertyId);

}

function getPropertiesByOwner(address owner) public view returns (uint256[] memory) {

return ownerProperties[owner];

}

}

Output:

To Execute the above code we need to give the address of the user contract as input:

User Contract: 0xB57ee0797C3fc0205714a577c02F7205bB89dF30

|  |  |
| --- | --- |
| **status** | 0x1 Transaction mined and execution succeed |
| **transaction hash** | 0x28b960c4659d6825b6eba79a1c434043c82ee7e2d49dccfd6022a843452b4a91 |
| **block hash** | 0x62003559802e0749e5873cecbd9e542fdb2191a2448eeaa17e6e309869784d2f |
| **block number** | 39 |
| **contract address** | 0x838F9b8228a5C95a7c431bcDAb58E289f5D2A4DC |
| **from** | 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 |
| **to** | PropertyContract.(constructor) |
| **gas** | 1325581 gas |
| **transaction cost** | 1152679 gas |
| **execution cost** | 1019149 gas |
| **input** | 0x608...9df30 |
| **decoded input** | { "address userContractAddress": "0xB57ee0797C3fc0205714a577c02F7205bB89dF30" } |
| **decoded output** | - |
| **logs** | [] |
| **raw logs** | [] |

3.Transaction Handling:

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

import "./PropertyContract.sol";

contract TransactionContract {

struct Transaction {

uint256 id;

uint256 propertyId;

address buyer;

address seller;

uint256 amount;

bool isCompleted;

bool isDisputed;

}

PropertyContract propertyContract;

uint256 public transactionCount;

mapping(uint256 => Transaction) public transactions;

mapping(address => uint256[]) public userTransactions;

event TransactionInitiated(uint256 transactionId, uint256 propertyId, address indexed buyer, address indexed seller, uint256 amount);

event TransactionCompleted(uint256 transactionId);

event TransactionDisputed(uint256 transactionId);

constructor(address propertyContractAddress) {

propertyContract = PropertyContract(propertyContractAddress);

}

function initiateTransaction(uint256 propertyId) public payable {

PropertyContract.Property memory property = propertyContract.properties(propertyId);

require(property.isListed, "Property is not listed for sale");

require(msg.value >= property.price, "Insufficient payment");

transactionCount++;

transactions[transactionCount] = Transaction({

id: transactionCount,

propertyId: propertyId,

buyer: msg.sender,

seller: property.owner,

amount: msg.value,

isCompleted: false,

isDisputed: false

});

userTransactions[msg.sender].push(transactionCount);

userTransactions[property.owner].push(transactionCount);

emit TransactionInitiated(transactionCount, propertyId, msg.sender, property.owner, msg.value);

}

function completeTransaction(uint256 transactionId) public {

Transaction storage transaction = transactions[transactionId];

require(transaction.buyer == msg.sender || transaction.seller == msg.sender, "Only buyer or seller can complete the transaction");

require(!transaction.isCompleted, "Transaction is already completed");

require(!transaction.isDisputed, "Transaction is disputed");

payable(transaction.seller).transfer(transaction.amount);

transaction.isCompleted = true;

emit TransactionCompleted(transactionId);

}

function raiseDispute(uint256 transactionId) public {

Transaction storage transaction = transactions[transactionId];

require(transaction.buyer == msg.sender || transaction.seller == msg.sender, "Only buyer or seller can raise a dispute");

require(!transaction.isCompleted, "Cannot dispute a completed transaction");

transaction.isDisputed = true;

emit TransactionDisputed(transactionId);

}

}

Output:

To execute the Transaction Contract we need to provide the address of the property contract:

Address: 0x838F9b8228a5C95a7c431bcDAb58E289f5D2A4DC

|  |  |
| --- | --- |
| **status** | 0x1 Transaction mined and execution succeed |
| **transaction hash** | 0x550ffd68d091eca170e59aafbc7846685ad7c3e10e96fff57dc471e20406d3db |
| **block hash** | 0x3cd372cfa46a79228517ee354c217c7a66f32136d7cec6b27da3792620164b8d |
| **block number** | 40 |
| **contract address** | 0x9a2E12340354d2532b4247da3704D2A5d73Bd189 |
| **from** | 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 |
| **to** | TransactionContract.(constructor) |
| **gas** | 1300918 gas |
| **transaction cost** | 1131233 gas |
| **execution cost** | 999929 gas |
| **input** | 0x608...2a4dc |
| **decoded input** | { "address propertyContractAddress": "0x838F9b8228a5C95a7c431bcDAb58E289f5D2A4DC" } |
| **decoded output** | - |
| **logs** | [] |
| **raw logs** | [] |

4.Document handling and verification:

Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract DocumentContract {

struct Document {

uint256 id;

address uploader;

string hash;

bool isVerified;

}

uint256 public documentCount;

mapping(uint256 => Document) public documents;

event DocumentUploaded(uint256 documentId, address indexed uploader, string hash);

event DocumentVerified(uint256 documentId);

function uploadDocument(string memory hash) public {

documentCount++;

documents[documentCount] = Document({

id: documentCount,

uploader: msg.sender,

hash: hash,

isVerified: false

});

emit DocumentUploaded(documentCount, msg.sender, hash);

}

function verifyDocument(uint256 documentId) public {

Document storage document = documents[documentId];

require(!document.isVerified, "Document is already verified");

document.isVerified = true;

emit DocumentVerified(documentId);

}

function isDocumentVerified(uint256 documentId) public view returns (bool) {

return documents[documentId].isVerified;

}

}

Output:

|  |  |
| --- | --- |
| **status** | 0x1 Transaction mined and execution succeed |
| **transaction hash** | 0x9f1a408f03c0273b3dc88e749eac25b08835f4e5d606e7769a685970e0263d53 |
| **block hash** | 0xf6923d68290747eaaf4289b1d63769a555f3d76ecf9ada508f49d21321ce198f |
| **block number** | 41 |
| **contract address** | 0x3c725134d74D5c45B4E4ABd2e5e2a109b5541288 |
| **from** | 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 |
| **to** | DocumentContract.(constructor) |
| **gas** | 780698 gas |
| **transaction cost** | 678867 gas |
| **execution cost** | 580611 gas |
| **input** | 0x608...a0033 |
| **decoded input** | {} |
| **decoded output** | - |
| **logs** | [] |
| **raw logs** | [] |

5.Payment processing:

It is integrated with Transaction Contract where payments are handled during the transaction process.

6.**Dispute Resolution**

The dispute resolution process is initiated within the TransactionContract, where disputes can be raised, and arbitration can be handled through additional functionality.

This code covers the essential features outlined in your updated requirement specifications. You can further enhance the system by adding more detailed validation, external oracle integration, and modularizing the dispute resolution process if needed.

Test Case:

**Case 1: Successful registration**

Step1: I have compiled the User registration then I have deployed the contract.

Step2: Then after successful completion of the contract is listed under the “Deployed contracts” .

Step3: Then in the deployed contracts section I have given my email address to register user .

Step4: Then when I clicked on the register user it gave me output as

|  |  |
| --- | --- |
| **status** | 0x1 Transaction mined and execution succeed |
| **transaction hash** | 0x2efd9d1d9c023f22469e26fa0f56dd5bcedcf7b65b5ad88c4aaed0b3ca991940 |
| **block hash** | 0x4ee816c5df14b2aa75fc8bfdd0b57ac5709d7338f82694e54046618a6fb37876 |
| **block number** | 43 |
| **from** | 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 |
| **to** | UserContract.registerUser(string) 0xDA07165D4f7c84EEEfa7a4Ff439e039B7925d3dF |
| **gas** | 106447 gas |
| **transaction cost** | 92562 gas |
| **execution cost** | 70778 gas |
| **input** | 0x704...00000 |
| **decoded input** | { "string email": "dubli.ramprasath@gmail.com" } |
| **decoded output** | {} |
| **logs** | [ { "from": "0xDA07165D4f7c84EEEfa7a4Ff439e039B7925d3dF", "topic": "0x48cac28ad4dc618e15f4c2dd5e97751182f166de97b25618318b2112aa951a2f", "event": "UserRegistered", "args": { "0": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4", "1": "dubli.ramprasath@gmail.com", "user": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4", "email": "dubli.ramprasath@gmail.com" } } ] |
| **raw logs** | [ { "logIndex": "0x1", "blockNumber": "0x2b", "blockHash": "0x4ee816c5df14b2aa75fc8bfdd0b57ac5709d7338f82694e54046618a6fb37876", "transactionHash": "0x2efd9d1d9c023f22469e26fa0f56dd5bcedcf7b65b5ad88c4aaed0b3ca991940", "transactionIndex": "0x0", "address": "0xDA07165D4f7c84EEEfa7a4Ff439e039B7925d3dF", "data": "0x0000000000000000000000000000000000000000000000000000000000000020000000000000000000000000000000000000000000000000000000000000001a6475626c692e72616d7072617361746840676d61696c2e636f6d000000000000", "topics": [ "0x48cac28ad4dc618e15f4c2dd5e97751182f166de97b25618318b2112aa951a2f", "0x0000000000000000000000005b38da6a701c568545dcfcb03fcb875f56beddc4" ] } ] |

Step5: Then to check whether provided user is registered or not I have given the address generated in previous output which is “0x5B38Da6a701c568545dCfcB03FcB875f56beddC4” and then I have clicked on user registered to check the status of registration then I got this output

|  |  |
| --- | --- |
| **from** | 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 |
| **to** | UserContract.isUserRegistered(address) 0xDA07165D4f7c84EEEfa7a4Ff439e039B7925d3dF |
| **execution cost** | 2854 gas (Cost only applies when called by a contract) |
| **input** | 0x163...eddc4 |
| **decoded input** | { "address userAddress": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4" } |
| **decoded output** | { "0": "bool: true" } |
| **logs** | [] |
| **raw logs** | [] |

**Case 2: User Registration with Already Registered Wallet Address**

Step1: I have used the same mail to register again then it gave me error as output stating that the user is registered already.

A screenshot of a computer screen

Description automatically generated

Output:

**[vm]**

**from:** 0x5B3...eddC4

**to:** UserContract.registerUser(string) 0xDA0...5d3dF

**value:** 0 wei

**data:** 0x704...00000

**logs:** 0

**hash:** 0x787...0d89f

**Debug**

transact to UserContract.registerUser errored: Error occurred: revert.

revert

The transaction has been reverted to the initial state.

Reason provided by the contract: "**User already registered**".

You may want to cautiously increase the gas limit if the transaction went out of gas.

**Case3: Invalid Verification Code**

Step1: Then to test my contract with invalid verification code I have updated my code for user contract with additional code for verification.

Updated Code:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract UserContract {

    struct User {

        address walletAddress;

        string email;

        bool isRegistered;

        string verificationCode;  // New field for verification code

    }

    mapping(address => User) public users;

    event UserRegistered(address indexed user, string email, string verificationCode);

    // Register user with email but mark them as not verified until the code is confirmed

    function registerUser(string memory email) public {

        require(!users[msg.sender].isRegistered, "User already registered");

        // Generate a mock verification code (e.g., "1234")

        string memory code = "1234";  // In a real scenario, this would be generated dynamically

        // Store the user with the generated verification code but set isRegistered to false

        users[msg.sender] = User({

            walletAddress: msg.sender,

            email: email,

            isRegistered: false,  // User is not fully registered until verified

            verificationCode: code

        });

        emit UserRegistered(msg.sender, email, code);

    }

    // Simulate sending a verification code to the user (this function would send the code in a real system)

    function sendVerificationCode() public view returns (string memory) {

        require(bytes(users[msg.sender].email).length > 0, "User is not registered");

        return users[msg.sender].verificationCode;  // In practice, you'd send this to the user's email

    }

    // Verify the user by matching the entered code with the stored code

    function verifyUser(string memory code) public {

        require(bytes(users[msg.sender].verificationCode).length > 0, "User is not registered");

        require(!users[msg.sender].isRegistered, "User is already verified");

        // Check if the entered code matches the stored verification code

        require(keccak256(abi.encodePacked(users[msg.sender].verificationCode)) == keccak256(abi.encodePacked(code)), "Invalid verification code");

        // Mark the user as verified

        users[msg.sender].isRegistered = true;

    }

    // Check if a user is registered (and verified)

    function isUserRegistered(address userAddress) public view returns (bool) {

        return users[userAddress].isRegistered;

    }

}

Step2:

To verify the generated address.

Step1: Check whether user is registered by providing address “0x5B38Da6a701c568545dCfcB03FcB875f56beddC4” we can get this address by either in register user output or while checking whether user is registered .

Step2: Then provide the address in the users tab and click on the user .

A screenshot of a computer

Description automatically generated

Step3: Then after clicking on the users it will provide the output as

A screenshot of a computer program

Description automatically generated

Where the isregistered: True represents the user is registered.

Step3: Now we are going to check another scenario with invalid user address.

Scenario1:

I have used the address as 99 and it provided output as

call to UserContract.users errored: Error encoding arguments: Error: invalid address (argument="address", value="99", code=INVALID\_ARGUMENT, version=address/5.7.0) (argument=null, value="99", code=INVALID\_ARGUMENT, version=abi/5.7.0)

Scenario 2:

To check the user we should only use the hexadecimal code as address now I am going to use this fake address “0xF2A1B3C4D5E6F7890ABCDEF1234567890ABCDEF1” and output is as follows.

A screenshot of a computer program

Description automatically generated

Step3: Then to check updated code along with verification code I have complied and deployed with updated code then I found that verify user and send verification code is added.

Step1: Registering user [dubli.ramprasath@gmail.com](mailto:dubli.ramprasath@gmail.com) .

Output:

A screenshot of a computer program

Description automatically generated

A close up of lines

Description automatically generatedA screenshot of a computer program

Description automatically generated

Step2: When I initiate transact.

A screenshot of a computer

Description automatically generated

Output:

A screen shot of a computer

Description automatically generated A screenshot of a computer program

Description automatically generated

Step3: When I registered the user it provided me the verification in output there we can clearly see the verification code is “1234” and now I ma going to verify user with wrong code “99”



It provided output as “Invalid verification code”.

A computer screen with text

Description automatically generated

  
  
for correct verification code it provides output as

A screenshot of a computer program

Description automatically generated

Step4: Next checking for “Is user registered ” where I have provided the address of the added user which is “0x5B38Da6a701c568545dCfcB03FcB875f56beddC4”

A computer screen with numbers and symbols

Description automatically generated

Output :

A screenshot of a computer program

Description automatically generated

Step5: Then when I have clicked on send verification code in output code I have received the verification code 1234 in my output.

Output:

A screenshot of a computer program

Description automatically generated

**Test Case 2: Property Listing Management:**

Test Case 2.1: Successfully List a Property

Step1: After successful compilation of Property contract and user contract also when we are done with registration of the user.

Step2: Then we can see the list of options available under property contract in deployed contract section.

Step3: These are the list of options available.

A screenshot of a computer

Description automatically generated

Step4: While deploying the property contract we will provide the address of the user contract.

Output of successful deployment:

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generatedoutput:

Step5: In property contract:

A screenshot of a computer

Description automatically generated

A computer screen shot of a program

Description automatically generatedA screenshot of a computer

Description automatically generatedOutput:

A close up of a screen

Description automatically generated with medium confidence

**Test Case 2.2:** **Update Property Details**

**Step1: For sample input I have given:**

**Property Id: 1**

**description: "Updated 2-bedroom apartment"**

**price: 1200.**

**Step2:**

**A screenshot of a computer

Description automatically generated**

**Output:**

**A screenshot of a computer program

Description automatically generated**

**A computer screen with text

Description automatically generated**

**Test Case 2.3: Remove Property from Listing**

Step1: To remove the property contract , I have typed in property ID as 1.

Step2: Then I have given the transact and it gave output as

A screenshot of a computer

Description automatically generated

A computer screen shot of text

Description automatically generated

**Test case 3: Transaction Handling**

**3.1: Successfully Initiate a Transaction**

Registering Seller:

Step1: As a first step I have checked and verified the user registration status by providing the already registered user’s address “0x5B38Da6a701c568545dCfcB03FcB875f56beddC4”.

Step2: Then provided me output as

A screenshot of a computer program

Description automatically generated

Verification:

A screenshot of a computer program

Description automatically generated

**A screenshot of a computer

Description automatically generated**

Step 3: Before registering seller I ma going to use different account in Remix IDE in which   
 the account which starts with “0x5b” is buyer account and “0xAb” is seller account.

Step4: Now I am going to register seller from different account

Seller mail ID: [seller@gmail.com](mailto:seller@gmail.com) .  
 This output ensures successful registration of seller.

Output:

A computer screen shot of text

Description automatically generated A screenshot of a computer

Description automatically generated

Step5: Verification of registered seller by providing verification “1234” .

A screenshot of a computer

Description automatically generatedOutput:

Step 6: Then to check is user registered I have provided the address of the user “0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2”

A screenshot of a computer program

Description automatically generated

Step7: Checking with different address “0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb1”

Output:

call to UserContract.isUserRegistered errored: Error encoding arguments: Error: bad address checksum (argument="address", value="0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb1", code=INVALID\_ARGUMENT, version=address/5.7.0) (argument=null, value="0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb1", code=INVALID\_ARGUMENT, version=abi/5.7.0)

Step8: While checking user with this address it will list the email Id status of registration and verification.

A screenshot of a computer screen

Description automatically generated

**Listing property from seller side:**

Step1: Then I have deployed the property contract with seller’s address.

Step2: After successful compilation and deployment it provided output as follows .

A screenshot of a computer

Description automatically generated

Step 3: Then to list the property I have given the   
 Description: “"1234 Blockchain Street, Crypto City, 12345"”  
 Price: 1000000000000000000

Output:

A screenshot of a computer program

Description automatically generated

A computer screen with text

Description automatically generated  
  
from the observation of above output the property ID is “2”.

Step4: Now to initiate the transaction from the buyer side switch to buyer account.