

Smart Wallet Requirements Analysis

User

A smart wallet is a next-generation digital wallet that combines traditional payment functionality with advanced blockchain technology and programmable features. Unlike basic cryptocurrency wallets that simply store and transfer digital assets, smart wallets can execute complex operations through smart contracts, automate transactions, and interact directly with decentralized applications. They offer enhanced security through features like multi-signature authentication, social recovery options, and customizable spending limits. Smart wallets also enable users to manage multiple cryptocurrencies, tokens, and NFTs in one interface, while providing advanced features like account abstraction, gas fee optimization, and batch transactions. This technology is revolutionizing how users interact with Web3 platforms by simplifying complex blockchain operations and offering a more user-friendly experience comparable to traditional financial services.

Identify functional and non-functional requirements from the system description.

ChatGPT

Functional Requirements

1. ****Multi-Cryptocurrency Management****: The system should support the storage and management of multiple cryptocurrencies, tokens, and NFTs.
2. ****Smart Contract Execution****: The wallet should enable the execution of complex operations via smart contracts.
3. ****Automated Transactions****: The wallet should support automation of transactions based on predefined criteria.
4. ****Integration with Decentralized Applications (dApps)****: The wallet should directly interact with decentralized applications.
5. ****Enhanced Security Mechanisms****:
 - ****Multi-Signature Authentication****: The system should support multi-signature authentication to enhance security.
 - ****Social Recovery Options****: The wallet should include options for social recovery to facilitate account recovery.

6. **Customizable Spending Limits**: Users should be able to set spending limits to control their transactions.
7. **Account Abstraction**: The wallet should support account abstraction, allowing simplified user interaction with blockchain operations.
8. **Gas Fee Optimization**: The system should have mechanisms to optimize transaction fees on the blockchain.
9. **Batch Transactions**: The wallet should support the batching of transactions to streamline multiple operations.

Non-Functional Requirements

1. **Usability**: The wallet interface should be user-friendly, making complex blockchain operations accessible to users.
2. **Performance**: Transaction processing should be optimized for efficiency, ensuring minimal delays for users.
3. **Reliability**: The wallet should consistently perform transactions and contract executions without errors.
4. **Scalability**: The wallet should handle an increasing number of users and transactions without compromising performance.
5. **Security**: The wallet should incorporate advanced security protocols to protect user assets and data.
6. **Interoperability**: The wallet should be compatible with various blockchain networks and standards.
7. **Maintainability**: The wallet software should be easy to update and maintain to address potential bugs, security patches, and feature enhancements.
8. **Availability**: The wallet service should be accessible 24/7, with minimal downtime.
9. **Compliance**: The wallet should comply with relevant legal and regulatory standards for digital asset management.

User

Create a use case model for the system.

ChatGPT

Here's a **use case model** for the **Smart Wallet System** based on its key functionalities:

Actors

1. **User**: Individual using the smart wallet for managing assets and interacting with dApps.
2. **Decentralized Applications (dApps)**: External applications on the blockchain that interact with the smart wallet.
3. **Blockchain Network**: The underlying blockchain network for transaction validation, gas fee handling, and smart contract execution.
4. **Recovery Contacts**: Trusted contacts used for social recovery.

Use Cases

1. **Manage Assets**

- **Description**: Allows users to store, view, and manage multiple cryptocurrencies, tokens, and NFTs.
- **Actors**: User
- **Precondition**: User has logged into the wallet.
- **Main Flow**:
 1. User opens the wallet interface.
 2. User views the current portfolio and balances.
 3. User can send or receive any supported asset.

- **Postcondition**: Asset management completed successfully.

2. **Execute Smart Contracts**

- **Description**: Enables users to interact with smart contracts for executing complex blockchain operations.
- **Actors**: User, Blockchain Network
- **Precondition**: User is logged in and has selected a contract to execute.
- **Main Flow**:
 1. User selects a smart contract and provides required inputs.
 2. Wallet sends the transaction to the blockchain for execution.
 3. Blockchain validates and executes the smart contract.
- **Postcondition**: Smart contract is executed, and the results are displayed to the user.

3. **Automate Transactions**

- **Description**: Users set up rules for automating transactions based on predefined criteria.
- **Actors**: User, Blockchain Network
- **Precondition**: User is logged in and has funds to automate transactions.
- **Main Flow**:
 1. User sets up automation criteria (e.g., recurring payments).
 2. Wallet monitors criteria and triggers transactions accordingly.
 3. Blockchain network validates and records each transaction.
- **Postcondition**: Transaction automation rules are active and execute as defined.

4. **Interact with dApps**

- **Description**: Users can connect their smart wallet to decentralized applications for various purposes (e.g., DeFi, NFTs).

- **Actors**: User, dApps
- **Precondition**: User is logged in and has accepted connection permissions.
- **Main Flow**:
 1. User connects the wallet to a dApp.
 2. User performs actions within the dApp (e.g., staking, trading).
 3. Wallet manages transactions between the user and dApp.
- **Postcondition**: dApp interaction is completed.

5. **Enhanced Security Features**

- **Description**: Provides additional security through multi-signature and customizable spending limits.
- **Actors**: User, Recovery Contacts
- **Precondition**: User has activated security features.
- **Main Flow**:
 1. User sets up multi-signature authentication and spending limits.
 2. Recovery contacts can be designated if social recovery is needed.
 3. User confirms security configurations.
- **Postcondition**: Enhanced security is active.

6. **Social Recovery**

- **Description**: Allows the user to recover account access through pre-assigned recovery contacts.
- **Actors**: User, Recovery Contacts
- **Precondition**: User has set up social recovery contacts.
- **Main Flow**:
 1. User initiates social recovery process.
 2. Recovery contacts approve the recovery request.

3. Account access is restored for the user.

- **Postcondition**: User regains access to the wallet.

7. **Optimize Gas Fees**

- **Description**: Manages gas fee costs by optimizing transaction timing and network usage.

- **Actors**: User, Blockchain Network

- **Precondition**: User is conducting a transaction on a network that supports gas optimization.

- **Main Flow**:

1. User initiates a transaction.

2. Wallet analyzes optimal times and methods to reduce fees.

3. Transaction is completed with minimized gas costs.

- **Postcondition**: Transaction is executed with optimized fees.

8. **Batch Transactions**

- **Description**: Allows users to bundle multiple transactions into a single batch.

- **Actors**: User, Blockchain Network

- **Precondition**: User has multiple transactions to complete.

- **Main Flow**:

1. User selects transactions to bundle.

2. Wallet combines transactions into a single batch.

3. Blockchain network processes the batch transaction.

- **Postcondition**: Batch transaction is executed successfully.

****Diagram of Use Cases****

For the above use cases, a visual use case diagram can be helpful in organizing actors and interactions systematically. Here's a breakdown:

- Place ****User**** at the center, with primary use cases (e.g., Manage Assets, Execute Smart Contracts, Automate Transactions) directly connected.
- ****Blockchain Network**** can be linked with use cases where network validation is required.
- ****dApps**** can be connected to ****Interact with dApps****.
- ****Recovery Contacts**** connect with ****Social Recovery**** for account access retrieval.

This covers essential interactions and system functionality at a high level, showcasing key workflows within the smart wallet.

User

Create use case specifications for identified use cases.

ChatGPT

Here are detailed ****use case specifications**** for the identified use cases for the ****Smart Wallet System****:

Use Case 1: ****Manage Assets****

- ****Description****: Enables users to store, view, send, and receive multiple cryptocurrencies, tokens, and NFTs.
- ****Actors****: User
- ****Preconditions****:

- User is authenticated and logged into the smart wallet.
- Wallet has access to the blockchain network.
- **Postconditions**: Assets are managed successfully, with updated balances.
- **Main Flow**:
 1. User accesses the "Assets" section in the wallet.
 2. User views their current portfolio, including balances and asset types.
 3. User selects an asset to send or receive.
 4. For sending, the user specifies the amount and recipient address, confirms, and the transaction is processed.
 5. For receiving, the user shares their wallet address, and incoming assets are automatically updated.
- **Alternative Flow**:
 - **Invalid Recipient Address**: If an invalid address is entered, the wallet displays an error message, prompting the user to correct the address.
- **Exceptions**:
 - **Network Failure**: If the network is down, the transaction is queued until connectivity is restored.

Use Case 2: **Execute Smart Contracts**

- **Description**: Allows users to execute complex blockchain operations via smart contracts.
- **Actors**: User, Blockchain Network
- **Preconditions**:
 - User is authenticated.

- User has sufficient balance to cover transaction fees and any smart contract costs.
- **Postconditions**: Smart contract operation is completed, and any resulting actions are updated in the wallet.
- **Main Flow**:
 1. User selects the smart contract feature and inputs relevant data for contract execution.
 2. Wallet processes the data and sends the transaction to the blockchain.
 3. Blockchain network validates and executes the smart contract.
 4. User receives confirmation, and the transaction details are recorded in their transaction history.
- **Alternative Flow**:
 - **Insufficient Funds**: If the user lacks funds for gas fees, an error message prompts the user to top up.
- **Exceptions**:
 - **Smart Contract Error**: If the contract fails during execution, an error message displays, and transaction details are reverted.

Use Case 3: **Automate Transactions**

- **Description**: Allows users to set up recurring or automated transactions based on specified criteria.
- **Actors**: User, Blockchain Network
- **Preconditions**: User has set automation criteria and sufficient funds.
- **Postconditions**: Automated transactions are executed according to defined criteria.
- **Main Flow**:
 1. User navigates to the automation feature.

2. User specifies criteria, such as frequency (e.g., daily, weekly), amount, and recipient.
 3. Wallet verifies criteria and initiates the automation setup.
 4. Blockchain network processes each automated transaction based on the schedule.
- **Alternative Flow**:
 - **Change in Criteria**: User can update or cancel automated criteria at any time.
 - **Exceptions**:
 - **Insufficient Funds**: If funds are insufficient for an automated transaction, the wallet notifies the user, and the transaction is suspended.

Use Case 4: **Interact with dApps**

- **Description**: Allows users to connect their wallet with decentralized applications for various activities like DeFi, gaming, and NFT marketplaces.
- **Actors**: User, dApps
- **Preconditions**: User grants permission for dApp access.
- **Postconditions**: User can seamlessly interact with dApps through the wallet.
- **Main Flow**:
 1. User selects the dApp to connect with.
 2. Wallet prompts for permissions; user confirms.
 3. User performs actions within the dApp, such as staking or trading.
 4. Wallet manages transactions and updates the user's balance based on dApp activity.
- **Alternative Flow**:
 - **Permission Denial**: If permissions are not granted, the dApp connection is not established.
- **Exceptions**:

- **dApp Connection Failure**: If the connection fails, the wallet notifies the user, and the interaction is canceled.

Use Case 5: **Enhanced Security Features**

- **Description**: Provides added security options, including multi-signature authentication and customizable spending limits.
- **Actors**: User, Recovery Contacts
- **Preconditions**: User has access to the wallet and has set up security options.
- **Postconditions**: Enhanced security features are active and functioning.
- **Main Flow**:
 1. User selects enhanced security options.
 2. User configures multi-signature authentication and sets spending limits.
 3. Wallet verifies and saves the security configurations.
- **Alternative Flow**:
 - **Configuration Change**: User can modify or disable security settings as needed.
- **Exceptions**:
 - **Unauthorized Access Attempt**: If an unauthorized access attempt is made, wallet triggers an alert and requires additional authentication.

Use Case 6: **Social Recovery**

- **Description**: Allows the user to regain wallet access using designated recovery contacts.
- **Actors**: User, Recovery Contacts
- **Preconditions**: Social recovery contacts have been previously set up.
- **Postconditions**: User regains wallet access.
- **Main Flow**:
 1. User initiates social recovery and notifies designated contacts.
 2. Recovery contacts approve the request, confirming their identities.
 3. Wallet restores access for the user.
- **Alternative Flow**:
 - **Delayed Response from Contacts**: Wallet sends reminders to contacts who have not responded.
- **Exceptions**:
 - **Contact Denial**: If recovery contacts deny the request, access is not restored, and the user must use alternative methods.

Use Case 7: **Optimize Gas Fees**

- **Description**: Optimizes transaction gas fees by choosing the best times and methods for processing transactions.
- **Actors**: User, Blockchain Network
- **Preconditions**: User is initiating a transaction on a blockchain network that supports gas fee optimization.
- **Postconditions**: Transaction is completed with optimized gas costs.
- **Main Flow**:

1. User initiates a transaction.
2. Wallet analyzes current network fees and optimizes timing and processing.
3. Transaction is sent with minimized gas costs.

- **Alternative Flow**:

- **Peak Network Usage**: Wallet suggests delaying the transaction until fees are lower.

- **Exceptions**:

- **Failed Optimization**: If optimization fails, the transaction proceeds with regular fees.

Use Case 8: **Batch Transactions**

- **Description**: Allows users to combine multiple transactions into one batch.

- **Actors**: User, Blockchain Network

- **Preconditions**: User has multiple pending transactions.

- **Postconditions**: Batch transaction is processed successfully.

- **Main Flow**:

1. User selects multiple transactions to batch.
2. Wallet creates a single batch transaction.
3. Blockchain processes the batch, and transaction details are recorded.

- **Alternative Flow**:

- **Exceeds Transaction Limit**: If the batch exceeds allowed transaction limits, user is prompted to reduce the batch size.

- **Exceptions**:

- **Batch Failure**: If the batch processing fails, the wallet retries or processes transactions individually.

These specifications cover each use case's interactions, flows, conditions, and exceptions, providing a detailed roadmap for implementing the smart wallet functionalities.

User

Create a domain model based on use case specifications.

ChatGPT

Here's a **Domain Model** for the **Smart Wallet System**, detailing the essential classes and their relationships based on the use case specifications.

Classes and Attributes

1. **User**

- **Attributes**:
 - userId: String
 - name: String
 - email: String
 - walletAddress: String
 - securitySettings: SecuritySettings
 - assets: List<Asset>
 - recoveryContacts: List<RecoveryContact>
- **Relationships**:
 - **1-to-1** with **Wallet**
 - **1-to-many** with **Asset**
 - **1-to-1** with **SecuritySettings**

- ****1-to-many**** with ****RecoveryContact****

2. ****Wallet****

- ****Attributes****:
 - walletId: String
 - userId: String (foreign key)
 - balance: BigDecimal
 - transactionHistory: List<Transaction>
 - smartContracts: List<SmartContract>
 - dAppConnections: List<dApp>
- ****Relationships****:
 - ****1-to-1**** with ****User****
 - ****1-to-many**** with ****Transaction****
 - ****1-to-many**** with ****SmartContract****
 - ****1-to-many**** with ****dApp****

3. ****Asset****

- ****Attributes****:
 - assetId: String
 - type: String (e.g., cryptocurrency, token, NFT)
 - balance: BigDecimal
 - blockchainNetwork: String
- ****Relationships****:
 - ****many-to-1**** with ****User****

4. ****SmartContract****

- ****Attributes****:
 - contractId: String
 - parameters: Map\<String, Object\>
 - executionStatus: String (e.g., pending, completed, failed)
 - gasFee: BigDecimal
 - transaction: Transaction
- ****Relationships****:
 - ****many-to-1**** with ****Wallet****
 - ****1-to-1**** with ****Transaction****

5. ****Transaction****

- ****Attributes****:
 - transactionId: String
 - transactionType: String (e.g., send, receive, automate, batch)
 - amount: BigDecimal
 - timestamp: Date
 - status: String (e.g., pending, completed)
 - gasFee: BigDecimal
 - batchId: String (nullable, if part of a batch transaction)
 - automated: Boolean
- ****Relationships****:
 - ****1-to-1**** with ****SmartContract****
 - ****many-to-1**** with ****Wallet****

6. ****dApp****

- ****Attributes****:

- dAppld: String
- name: String
- permissions: List\<String\>
- connectedStatus: Boolean
- lastInteractionDate: Date
- **Relationships**:
 - **many-to-1** with **Wallet**

7. **SecuritySettings**

- **Attributes**:
 - settingsId: String
 - multiSigEnabled: Boolean
 - spendingLimit: BigDecimal
 - recoveryEnabled: Boolean
- **Relationships**:
 - **1-to-1** with **User**

8. **RecoveryContact**

- **Attributes**:
 - contactId: String
 - name: String
 - contactMethod: String (e.g., email, phone)
 - status: String (e.g., pending, confirmed)
- **Relationships**:
 - **many-to-1** with **User**

9. **BlockchainNetwork**

- **Attributes**:
 - networkId: String
 - name: String
 - optimizationEnabled: Boolean
- **Relationships**:
 - **1-to-many** with **Transaction**
 - **1-to-many** with **Asset**

10. **BatchTransaction**

- **Attributes**:
 - batchId: String
 - transactions: List<Transaction>
 - status: String (e.g., pending, completed, failed)
- **Relationships**:
 - **1-to-many** with **Transaction**

Relationships Overview

- **User and Wallet**: Each user has one wallet.
- **User and Asset**: A user may own multiple assets (cryptocurrencies, tokens, NFTs).
- **Wallet and Transaction**: Wallet maintains a transaction history; a transaction can represent different types like sending/receiving funds, automated transactions, or batch transactions.

- **Wallet and SmartContract**: Wallets interact with multiple smart contracts for complex operations.
- **Wallet and dApp**: A wallet can connect to multiple dApps for interaction.
- **User and SecuritySettings**: Each user has security settings for multi-signature and spending limits.
- **User and RecoveryContact**: Users have multiple recovery contacts for account recovery.
- **Transaction and BlockchainNetwork**: Each transaction is validated on a blockchain network, which also supports gas optimization.
- **BatchTransaction and Transaction**: A batch transaction groups multiple individual transactions.

Domain Model Diagram Summary

This model visually represents the structure and relationships in the **Smart Wallet System**:

- **Entities**: The core entities like **User**, **Wallet**, **Asset**, **Transaction**, **SmartContract**, **dApp**, **SecuritySettings**, and **RecoveryContact**.
- **Relationships**: Displays relationships and multiplicities (e.g., **1-to-many** between User and Asset, **1-to-1** between User and Wallet).
- **Attributes**: Each class includes key attributes necessary for operations like managing assets, interacting with smart contracts and dApps, automating transactions, and providing security and recovery features.

This domain model provides a structured overview, supporting functionality requirements while maintaining modularity and reusability across the smart wallet's core functions.

User

Identify system operations from use case specifications.

ChatGPT

Here are the **system operations** identified from the use case specifications for the **Smart Wallet System**. These operations align with the key functionalities required to support the use cases.

1. **Manage Assets**

- **viewAssets(userId)**: Retrieves and displays the list of assets and their balances for the given user.
- **sendAsset(userId, assetId, recipientAddress, amount)**: Sends the specified amount of an asset to the recipient's address.
- **receiveAsset(userId, assetId, senderAddress, amount)**: Records the receipt of an asset from a sender's address and updates the user's balance.
- **updateAssetBalance(userId, assetId)**: Refreshes the balance of an asset in the user's wallet.

2. **Execute Smart Contracts**

- **executeSmartContract(userId, contractId, parameters)**: Executes the smart contract with the specified parameters.
- **verifyContractExecution(transactionId)**: Checks the status of a smart contract execution to confirm if it succeeded or failed.
- **recordContractTransaction(userId, transactionId)**: Logs a transaction resulting from smart contract execution in the user's transaction history.

3. ****Automate Transactions****

- ****setAutomationRules(userId, criteria)****: Defines criteria for transaction automation (e.g., schedule, amount, recipient).
- ****triggerAutomatedTransaction(userId, transactionDetails)****: Initiates an automated transaction based on the predefined criteria.
- ****suspendAutomatedTransaction(userId, transactionId)****: Pauses an automated transaction if conditions (e.g., insufficient funds) are not met.
- ****updateAutomationCriteria(userId, newCriteria)****: Allows the user to modify existing automation criteria.

4. ****Interact with dApps****

- ****connectToDApp(userId, dAppId)****: Connects the user's wallet to the specified dApp.
- ****authorizeDAppPermissions(userId, dAppId, permissions)****: Sets permissions for the dApp to access specific wallet functions.
- ****disconnectFromDApp(userId, dAppId)****: Disconnects the wallet from a dApp and revokes permissions.
- ****recordDAppInteraction(userId, dAppId, action)****: Logs an interaction with a dApp, such as staking or trading, in the transaction history.

5. ****Enhanced Security Features****

- ****setMultiSignature(userId, settings)****: Configures multi-signature requirements for enhanced security.

- **setSpendingLimit(userId, limit)**: Defines spending limits to control wallet transactions.
- **authenticateMultiSignature(userId, transactionId, signatures)**: Verifies multiple signatures for transaction approval.
- **configureRecoveryContacts(userId, contacts)**: Sets up contacts for social recovery in case of account lockout.

6. **Social Recovery**

- **initiateSocialRecovery(userId)**: Starts the social recovery process if the user cannot access their account.
- **notifyRecoveryContacts(userId, recoveryRequest)**: Sends notifications to recovery contacts requesting account recovery approval.
- **approveRecoveryRequest(contactId, userId)**: Allows a recovery contact to approve the social recovery request.
- **restoreAccountAccess(userId)**: Restores access to the user's account after sufficient recovery contact approvals are obtained.

7. **Optimize Gas Fees**

- **analyzeGasFees(transactionId)**: Evaluates the current network gas fees to determine the optimal time and method for transaction execution.
- **applyGasOptimization(transactionId)**: Adjusts transaction parameters to minimize gas costs based on network analysis.
- **suggestTransactionDelay(transactionId)**: Recommends delaying the transaction if fees are high and optimization cannot reduce costs effectively.

8. **Batch Transactions**

- **createBatchTransaction(userId, transactionList)**: Combines multiple individual transactions into a single batch for processing.
- **executeBatchTransaction(batchId)**: Initiates the processing of a batch transaction on the blockchain.
- **verifyBatchTransactionStatus(batchId)**: Checks the completion status of a batch transaction.
- **recordBatchTransaction(userId, batchId)**: Logs details of the batch transaction in the user's transaction history.

Summary of System Operations by Functionality

Functionality	Key System Operations
-----	-----
-----	-----
Manage Assets	viewAssets, sendAsset, receiveAsset, updateAssetBalance
Execute Smart Contracts	executeSmartContract, verifyContractExecution, recordContractTransaction
Automate Transactions	setAutomationRules, triggerAutomatedTransaction, suspendAutomatedTransaction, updateAutomationCriteria
Interact with dApps	connectToDApp, authorizeDAppPermissions, disconnectFromDApp, recordDAppInteraction
Enhanced Security	setMultiSignature, setSpendingLimit, authenticateMultiSignature, configureRecoveryContacts

Social Recovery	initiateSocialRecovery, notifyRecoveryContacts,
approveRecoveryRequest, restoreAccountAccess	
Optimize Gas Fees	analyzeGasFees, applyGasOptimization,
suggestTransactionDelay	
Batch Transactions	createBatchTransaction, executeBatchTransaction,
verifyBatchTransactionStatus, recordBatchTransaction	

These system operations facilitate smooth interactions with the smart wallet, aligning directly with the use case functionalities for managing assets, security, and transaction processing.

User

Create design sequence diagrams for system operations.

ChatGPT

Here are design sequence diagrams for some key ****system operations**** based on the Smart Wallet System functionality.

****1. Sequence Diagram for `sendAsset` Operation****

****Description****: This operation allows the user to send a specified asset to a recipient address.

****Actors****: User, WalletService, BlockchainNetwork, TransactionService

```plaintext

User -> WalletService: initiate sendAsset(assetId, recipientAddress, amount)

WalletService -> SecurityService: verifyMultiSignature(userId, assetId, amount)



```

SecurityService -> WalletService: authenticationSuccess()
alt authenticationSuccess
 WalletService -> BlockchainNetwork: createTransaction(assetId, recipientAddress,
amount)
 BlockchainNetwork -> WalletService: transactionConfirmation(transactionId)
 WalletService -> TransactionService: recordTransaction(userId, transactionId, status)
 TransactionService -> WalletService: transactionRecorded
 WalletService -> User: sendAssetConfirmation(transactionId)
else authenticationFailure
 WalletService -> User: authenticationError
end
` ` `

```

### \*\*2. Sequence Diagram for `executeSmartContract` Operation\*\*

#### \*\*Description\*\*: This operation executes a smart contract with specified parameters.

**Actors**: User, WalletService, BlockchainNetwork, TransactionService

` ` ` plaintext

```

User -> WalletService: executeSmartContract(contractId, parameters)
WalletService -> SecurityService: verifyMultiSignature(userId, contractId)
SecurityService -> WalletService: authenticationSuccess()
alt authenticationSuccess

```

```

 WalletService -> BlockchainNetwork: executeContract(contractId, parameters)

 BlockchainNetwork -> WalletService: contractExecutionResult(transactionId,
resultStatus)

 WalletService -> TransactionService: recordContractTransaction(userId, transactionId,
resultStatus)

 TransactionService -> WalletService: transactionRecorded

 WalletService -> User: contractExecutionConfirmation(resultStatus)
else authenticationFailure

 WalletService -> User: authenticationError
end
` ` `

```

### \*\*3. Sequence Diagram for `setAutomationRules` Operation\*\*

#### \*\*Description\*\*: This operation sets up automated transaction criteria, such as frequency and recipient.

\*\*Actors\*\*: User, WalletService, AutomationService

` ` ` plaintext

```
User -> WalletService: setAutomationRules(criteria)
```

```
WalletService -> SecurityService: verifyUserAuthorization(userId)
```

```
SecurityService -> WalletService: authorizationSuccess()
```

```
alt authorizationSuccess
```

```
 WalletService -> AutomationService: defineAutomationRules(userId, criteria)
```

AutomationService -> WalletService: automationRulesSet

WalletService -> User: automationConfirmation

else authorizationFailure

WalletService -> User: authorizationError

end

```\n

---\n

4. Sequence Diagram for `initiateSocialRecovery` Operation

Description: This operation allows the user to initiate social recovery to regain access to their account.

Actors: User, WalletService, SecurityService, RecoveryContacts

```\nplaintext

User -> WalletService: initiateSocialRecovery(userId)

WalletService -> SecurityService: validateRecoverySetup(userId)

SecurityService -> WalletService: setupConfirmed

alt setupConfirmed

WalletService -> RecoveryContacts: notifyRecoveryRequest(userId)

loop notifyEachContact

RecoveryContacts -> WalletService: recoveryApproval(contactId, userId)

end

WalletService -> SecurityService: confirmRecoveryApprovals(userId)

SecurityService -> WalletService: approvalsVerified

WalletService -> User: accountAccessRestored

else setupNotConfirmed

WalletService -> User: recoverySetupError

end

```\n

---\n

5. Sequence Diagram for `optimizeGasFees` Operation

Description: This operation analyzes and optimizes gas fees for a transaction.

Actors: User, WalletService, GasOptimizationService, BlockchainNetwork

```plaintext

User -> WalletService: initiateTransaction(transactionId)

WalletService -> GasOptimizationService: analyzeGasFees(transactionId)

GasOptimizationService -> WalletService: gasFeeSuggestion(optimalFee)

alt optimalFeeAvailable

WalletService -> BlockchainNetwork: executeTransaction(transactionId, optimalFee)

BlockchainNetwork -> WalletService: transactionConfirmation(transactionId, finalFee)

WalletService -> User: transactionConfirmation(finalFee)

else noOptimalFee

WalletService -> User: suggestTransactionDelay

end

` ` `

---

### \*\*6. Sequence Diagram for `createBatchTransaction` Operation\*\*

#### \*\*Description\*\*: This operation allows the user to batch multiple transactions into one and send it to the network.

\*\*Actors\*\*: User, WalletService, BlockchainNetwork, TransactionService

` ` ` plaintext

User -> WalletService: createBatchTransaction(transactionList)

WalletService -> SecurityService: verifyUserAuthorization(userId)

SecurityService -> WalletService: authorizationSuccess()

alt authorizationSuccess

    WalletService -> BlockchainNetwork: processBatchTransaction(transactionList)

    BlockchainNetwork -> WalletService: batchTransactionConfirmation(batchId)

    WalletService -> TransactionService: recordBatchTransaction(userId, batchId)

    TransactionService -> WalletService: batchTransactionRecorded

    WalletService -> User: batchTransactionConfirmation(batchId)

else authorizationFailure

    WalletService -> User: authorizationError

end

` ` `

---

These sequence diagrams illustrate the flow of interactions between various components of the Smart Wallet System, ensuring that each operation handles security checks, authorization, blockchain interaction, and user notifications in an organized and secure manner.

## User

Create design class diagrams based on the domain model and sequence diagrams.

## ChatGPT

Below is a **design class diagram** based on the domain model and the sequence diagrams provided. This diagram illustrates the classes, their attributes, methods, and relationships within the **Smart Wallet System**.

---

### ### **Design Class Diagram**

#### 1. **User**

##### - **Attributes**:

- userId: String
- name: String
- email: String
- walletAddress: String

##### - **Methods**:

- initiateSendAsset(assetId, recipientAddress, amount)
- initiateSmartContract(contractId, parameters)
- initiateSocialRecovery()

- setAutomationRules(criteria)

## 2. **Wallet**

- **Attributes**:
  - walletId: String
  - balance: BigDecimal
  - transactionHistory: List<Transaction>
- **Methods**:
  - viewAssets()
  - sendAsset(assetId, recipientAddress, amount)
  - executeSmartContract(contractId, parameters)
  - createBatchTransaction(transactionList)
  - connectToDApp(dAppId)
  - authorizeDAppPermissions(dAppId, permissions)
  - disconnectFromDApp(dAppId)

## 3. **Asset**

- **Attributes**:
  - assetId: String
  - type: String
  - balance: BigDecimal
  - blockchainNetwork: String
- **Methods**:
  - updateBalance()
  - getBalance()

#### 4. **SmartContract**

- **Attributes**:
  - contractId: String
  - parameters: Map\<String, Object\>
  - executionStatus: String
- **Methods**:
  - execute(parameters)

#### 5. **Transaction**

- **Attributes**:
  - transactionId: String
  - transactionType: String
  - amount: BigDecimal
  - timestamp: Date
  - status: String
  - gasFee: BigDecimal
- **Methods**:
  - processTransaction()
  - verifyTransactionStatus()

#### 6. **BatchTransaction**

- **Attributes**:
  - batchId: String
  - transactions: List\<Transaction\>
  - status: String
- **Methods**:



- executeBatch()
- verifyBatchStatus()

## 7. **dApp**

- **Attributes**:
  - dAppId: String
  - name: String
  - permissions: List<String>
  - connectedStatus: Boolean
- **Methods**:
  - connect()
  - authorizePermissions(permissions)
  - disconnect()

## 8. **SecuritySettings**

- **Attributes**:
  - settingsId: String
  - multiSigEnabled: Boolean
  - spendingLimit: BigDecimal
- **Methods**:
  - setMultiSignature(enabled)
  - setSpendingLimit(limit)
  - verifyMultiSignature(signatures)

## 9. **RecoveryContact**

- **Attributes**:

- contactId: String
- name: String
- contactMethod: String
- status: String
- **Methods**:
  - approveRecoveryRequest(userId)

#### 10. **BlockchainNetwork**

- **Attributes**:
  - networkId: String
  - name: String
  - optimizationEnabled: Boolean
- **Methods**:
  - executeTransaction(transactionId, fee)
  - executeContract(contractId, parameters)
  - verifyTransaction(transactionId)

#### 11. **GasOptimizationService**

- **Attributes**:
  - optimalFee: BigDecimal
- **Methods**:
  - analyzeGasFees(transactionId)
  - applyOptimization(transactionId)

#### 12. **AutomationService**

- **Attributes**:

- criteria: Map\<String, Object\>
- **\*\*Methods\*\***:
  - defineAutomationRules(userId, criteria)
  - triggerAutomatedTransaction(userId, transactionDetails)
  - suspendTransaction(userId, transactionId)

### 13. **\*\*SecurityService\*\***

- **\*\*Attributes\*\***:
  - authenticationStatus: Boolean
- **\*\*Methods\*\***:
  - verifyMultiSignature(userId, assetId, amount)
  - authenticateUser(userId)
  - validateRecoverySetup(userId)

### 14. **\*\*TransactionService\*\***

- **\*\*Attributes\*\***:
  - transactionRecords: List\<Transaction\>
- **\*\*Methods\*\***:
  - recordTransaction(userId, transactionId, status)
  - recordBatchTransaction(userId, batchId)

---

### ### **\*\*Class Relationships\*\***

- **\*\*User\*\*** ↔ **\*\*Wallet\*\***: 1-to-1 (Each user has one wallet)

- **User** ↔ **SecuritySettings**: 1-to-1 (User has one security settings configuration)
- **User** ↔ **RecoveryContact**: 1-to-many (User may have multiple recovery contacts)
- **Wallet** ↔ **Asset**: 1-to-many (Wallet can manage multiple assets)
- **Wallet** ↔ **Transaction**: 1-to-many (Wallet has a transaction history)
- **Wallet** ↔ **SmartContract**: 1-to-many (Wallet may execute multiple smart contracts)
- **Wallet** ↔ **dApp**: 1-to-many (Wallet can connect to multiple dApps)
- **Transaction** ↔ **BlockchainNetwork**: many-to-1 (Transactions are executed on a blockchain network)
- **BatchTransaction** ↔ **Transaction**: 1-to-many (Batch transactions contain multiple transactions)
- **GasOptimizationService** ↔ **Transaction**: 1-to-1 (Service optimizes each transaction's gas fee)
- **AutomationService** ↔ **Transaction**: 1-to-many (Service triggers automated transactions)
- **SecurityService** ↔ **User**: 1-to-1 (Service verifies user authentication and multi-signature)
- **TransactionService** ↔ **Transaction**: 1-to-many (Service records and manages user transactions)

---

This **class diagram** reflects the structure and interactions within the Smart Wallet System, ensuring it can support asset management, smart contract execution, automated transactions, security features, and optimized gas fees while interacting with blockchain networks and dApps. Each class's methods and attributes are defined to align with system operations and enhance the wallet's functionality and security.

## User

Develop a Java Implementation for the system as specified in the class diagram and sequence diagrams.

## ChatGPT

To implement the **Smart Wallet System** in Java, I'll outline the primary classes, attributes, and methods according to the class diagram and the operations from the sequence diagrams. For brevity, I'll provide a high-level implementation with key methods and some essential classes. You can expand on this by implementing additional methods and services.

Here's an implementation based on the design specified:

### 1. **User.java**

```
```java
```

```
import java.util.List;
```

```
public class User {
```

```
    private String userId;
```

```
    private String name;
```

```
    private String email;
```

```
    private String walletAddress;
```

```
    private Wallet wallet;
```

```
    private SecuritySettings securitySettings;
```

```
    private List<RecoveryContact> recoveryContacts;
```

```
    public User(String userId, String name, String email, String walletAddress) {
```

```
        this.userId = userId;
```

```
        this.name = name;
```

```
        this.email = email;
```

```

        this.walletAddress = walletAddress;
    }

    public void initiateSendAsset(String assetId, String recipientAddress, double amount) {
        wallet.sendAsset(assetId, recipientAddress, amount);
    }

    public void initiateSmartContract(String contractId, String parameters) {
        wallet.executeSmartContract(contractId, parameters);
    }

    public void initiateSocialRecovery() {
        securitySettings.initiateSocialRecovery(this);
    }

    public void setAutomationRules(String criteria) {
        wallet.setAutomationRules(criteria);
    }
}

...

---

### 2. **Wallet.java**

```java

```

```
import java.util.List;

public class Wallet {

 private String walletId;

 private double balance;

 private List<Transaction> transactionHistory;

 private List<Asset> assets;

 private List<dApp> dApps;

 public void viewAssets() {

 for (Asset asset : assets) {

 System.out.println(asset.getBalance());

 }

 }

 public void sendAsset(String assetId, String recipientAddress, double amount) {

 SecurityService securityService = new SecurityService();

 boolean authenticated = securityService.verifyMultiSignature(this.walletId, assetId,
amount);

 if (authenticated) {

 TransactionService transactionService = new TransactionService();

 String transactionId = transactionService.processTransaction(assetId,
recipientAddress, amount);

 transactionService.recordTransaction(this.walletId, transactionId, "Completed");

 } else {

 System.out.println("Authentication failed.");

 }

 }

}
```

```

 }

 public void executeSmartContract(String contractId, String parameters) {
 SecurityService securityService = new SecurityService();
 if (securityService.authenticateUser(this.walletId)) {
 TransactionService transactionService = new TransactionService();
 String transactionId = transactionService.executeContract(contractId, parameters);
 transactionService.recordContractTransaction(this.walletId, transactionId,
"Completed");
 } else {
 System.out.println("Smart contract execution failed due to authentication.");
 }
 }
}

```

```

 public void setAutomationRules(String criteria) {
 AutomationService automationService = new AutomationService();
 automationService.defineAutomationRules(this.walletId, criteria);
 }
}
...

```

---

### 3. \*\*Asset.java\*\*

```

```java

```



```

public class Asset {
    private String assetId;
    private String type;
    private double balance;
    private String blockchainNetwork;

    public Asset(String assetId, String type, double balance, String blockchainNetwork) {
        this.assetId = assetId;
        this.type = type;
        this.balance = balance;
        this.blockchainNetwork = blockchainNetwork;
    }

    public double getBalance() {
        return this.balance;
    }

    public void updateBalance(double newBalance) {
        this.balance = newBalance;
    }
}

...

---
```

4. **Transaction.java**

```
```java
```

```
import java.util.Date;
```

```
public class Transaction {
```

```
 private String transactionId;
```

```
 private String transactionType;
```

```
 private double amount;
```

```
 private Date timestamp;
```

```
 private String status;
```

```
 private double gasFee;
```

```
 public Transaction(String transactionId, String transactionType, double amount) {
```

```
 this.transactionId = transactionId;
```

```
 this.transactionType = transactionType;
```

```
 this.amount = amount;
```

```
 this.timestamp = new Date();
```

```
 this.status = "Pending";
```

```
 }
```

```
 public void processTransaction() {
```

```
 System.out.println("Processing transaction...");
```

```
 this.status = "Completed";
```

```
 }
```

```
 public void verifyTransactionStatus() {
```

```
 System.out.println("Transaction status: " + this.status);
 }
}
```

---
```

5. ****SecuritySettings.java****

```
```java
public class SecuritySettings {
 private boolean multiSigEnabled;
 private double spendingLimit;

 public void setMultiSignature(boolean enabled) {
 this.multiSigEnabled = enabled;
 }

 public void setSpendingLimit(double limit) {
 this.spendingLimit = limit;
 }

 public boolean initiateSocialRecovery(User user) {
 System.out.println("Initiating social recovery for user: " + user.getUserId());
 return true;
 }
}
```

```

public boolean verifyMultiSignature(String walletId, String assetId, double amount) {
 if (multiSigEnabled) {
 System.out.println("Multi-signature verification required.");
 // Verification logic here
 return true;
 }
 return false;
}
}
` ``

```

---

### 6. \*\*TransactionService.java\*\*

` `` java

```

public class TransactionService {
 public String processTransaction(String assetId, String recipientAddress, double amount)
 {
 Transaction transaction = new Transaction("TX123", "send", amount);
 transaction.processTransaction();
 return transaction.getTransactionId();
 }

 public void recordTransaction(String walletId, String transactionId, String status) {

```

```

 System.out.println("Recording transaction for wallet: " + walletId + " with status: " +
status);
 }

```

```

public String executeContract(String contractId, String parameters) {
 System.out.println("Executing smart contract: " + contractId);
 return "TX_CONTRACT123";
}

```

```

public void recordContractTransaction(String walletId, String transactionId, String status)
{
 System.out.println("Recording smart contract transaction with status: " + status);
}
}
` ` `

```

---

### 7. \*\*dApp.java\*\*

` `` `java

```
import java.util.List;
```

```

public class dApp {
 private String dAppId;
 private String name;
 private List<String> permissions;
}

```

```
private boolean connectedStatus;
```

```
public void connect() {
```

```
 this.connectedStatus = true;
```

```
 System.out.println("Connected to dApp: " + this.name);
```

```
}
```

```
public void authorizePermissions(List<String> permissions) {
```

```
 this.permissions = permissions;
```

```
}
```

```
public void disconnect() {
```

```
 this.connectedStatus = false;
```

```
 System.out.println("Disconnected from dApp: " + this.name);
```

```
}
```

```
}
```

```
```\n
```

```
---
```

```
### **Main.java** (Example of Using the System)
```

```
```java
```

```
public class Main {
```

```
 public static void main(String[] args) {
```

```
 User user = new User("U001", "Alice", "alice@example.com", "0xWalletAddress123");
```

```

Wallet wallet = new Wallet();

// Example operation: Sending an asset
user.initiateSendAsset("BTC", "0xRecipientAddress", 0.5);

// Example operation: Executing a smart contract
user.initiateSmartContract("SC001", "{data}");

// Example operation: Setting up automation rules
user.setAutomationRules("{frequency: 'daily', amount: 1}");

// Example operation: Initiating social recovery
user.initiateSocialRecovery();
}
}
...

```

This Java code provides a basic implementation with the following components:

- **User** initiates actions such as sending assets, executing smart contracts, and starting social recovery.
- **Wallet** manages assets and interacts with services to process transactions and automation rules.
- **TransactionService** and **SecurityService** handle transaction processing and multi-signature verification.
- **dApp** allows wallet connections and disconnections with decentralized applications.

Each class is structured to match the class diagram, supporting the core operations of the Smart Wallet System. Additional methods and services can be implemented to extend functionality further.

## **User**

Develop tests including unit tests, integration tests, and system tests for the implementation.

## **ChatGPT**

Here's a set of test cases that include **unit tests**, **integration tests**, and **system tests** for the **Smart Wallet System** implementation.

### **### 1. Unit Tests**

Unit tests focus on testing individual components and methods. We'll use JUnit, a common Java testing framework.

#### **#### Unit Tests for User Class**

```
```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class UserTest {
    private User user;
    private Wallet wallet;
```


@BeforeEach

```
public void setUp() {
```

```
    wallet = new Wallet("Wallet123", 1000.0);
```

```
    user = new User("U001", "Alice", "alice@example.com", "0xWalletAddress123", wallet);
```

```
}
```

@Test

```
public void testInitiateSendAsset() {
```

```
    user.initiateSendAsset("BTC", "0xRecipientAddress", 0.5);
```

```
    assertEquals(999.5, wallet.getBalance(), "Wallet balance should be updated after  
sending asset");
```

```
}
```

@Test

```
public void testInitiateSmartContract() {
```

```
    user.initiateSmartContract("SC001", "{parameters}");
```

```
    assertTrue(wallet.getTransactionHistory().size() > 0, "Transaction history should  
contain the smart contract execution record");
```

```
}
```

@Test

```
public void testInitiateSocialRecovery() {
```

```
    boolean result = user.initiateSocialRecovery();
```

```
    assertTrue(result, "Social recovery should be initiated successfully");
```

```
}
```

@Test

```

    public void testSetAutomationRules() {
        user.setAutomationRules("{frequency: 'daily', amount: 1}");
        assertNotNull(wallet.getAutomationRules(), "Automation rules should be set");
    }
}
```

```

#### \*\*Unit Tests for Wallet Class\*\*

```

```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class WalletTest {

    private Wallet wallet;

    @BeforeEach
    public void setUp() {
        wallet = new Wallet("Wallet123", 1000.0);
    }

    @Test
    public void testSendAsset() {
        wallet.sendAsset("BTC", "0xRecipientAddress", 0.5);
    }
}
```

```

```
 assertEquals(999.5, wallet.getBalance(), "Balance should decrease after sending
asset");
 }
}
```

```
@Test
public void testExecuteSmartContract() {
 wallet.executeSmartContract("SC001", "{parameters}");

 assertEquals(1, wallet.getTransactionHistory().size(), "Transaction history should have
a record of the smart contract execution");
}
}
```

```
@Test
public void testCreateBatchTransaction() {
 wallet.createBatchTransaction(List.of("TX001", "TX002"));

 assertEquals(1, wallet.getBatchTransactionHistory().size(), "Batch transaction should
be recorded");
}
}
...
}
```

#### \*\*Unit Tests for Transaction Class\*\*

```
```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
```

```

public class TransactionTest {

    private Transaction transaction;

    @BeforeEach

    public void setUp() {

        transaction = new Transaction("TX123", "send", 0.5);

    }

    @Test

    public void testProcessTransaction() {

        transaction.processTransaction();

        assertEquals("Completed", transaction.getStatus(), "Transaction status should be
completed");

    }

    @Test

    public void testVerifyTransactionStatus() {

        transaction.processTransaction();

        transaction.verifyTransactionStatus();

        assertEquals("Completed", transaction.getStatus(), "Transaction status should be
verified as completed");

    }

}

...

---
```

2. ****Integration Tests****

Integration tests verify that components work together as expected.

****Integration Test for User Sending Asset through Wallet****

```
```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class UserWalletIntegrationTest {

 private User user;

 private Wallet wallet;

 @BeforeEach

 public void setUp() {

 wallet = new Wallet("Wallet123", 1000.0);

 user = new User("U001", "Alice", "alice@example.com", "0xWalletAddress123", wallet);

 }

 @Test

 public void testSendAssetThroughWallet() {

 user.initiateSendAsset("BTC", "0xRecipientAddress", 0.5);

 assertEquals(999.5, wallet.getBalance(), "Balance should be deducted in wallet after sending asset");
 }
}
```

```
 assertTrue(wallet.getTransactionHistory().size() > 0, "Transaction should be recorded in
the wallet");
```

```
 }
```

```
}
```

```
```
```

```
#### **Integration Test for Smart Contract Execution**
```

```
```java
```

```
import static org.junit.jupiter.api.Assertions.*;
```

```
import org.junit.jupiter.api.BeforeEach;
```

```
import org.junit.jupiter.api.Test;
```

```
public class SmartContractIntegrationTest {
```

```
 private User user;
```

```
 private Wallet wallet;
```

```
 @BeforeEach
```

```
 public void setUp() {
```

```
 wallet = new Wallet("Wallet123", 1000.0);
```

```
 user = new User("U001", "Alice", "alice@example.com", "0xWalletAddress123", wallet);
```

```
 }
```

```
 @Test
```

```
 public void testExecuteSmartContract() {
```

```
 user.initiateSmartContract("SC001", "{parameters}");
```

```
 assertTrue(wallet.getTransactionHistory().size() > 0, "Smart contract execution should
be recorded in transaction history");
```

```
 }
```

```
}
```

```
...
```

```

```

### ### 3. **\*\*System Tests\*\***

System tests check the end-to-end functionality of the application.

#### #### **\*\*System Test: End-to-End Asset Transfer\*\***

```
```java
```

```
import static org.junit.jupiter.api.Assertions.*;
```

```
import org.junit.jupiter.api.BeforeEach;
```

```
import org.junit.jupiter.api.Test;
```

```
public class SmartWalletSystemTest {
```

```
    private User user;
```

```
    private Wallet wallet;
```

```
    private TransactionService transactionService;
```

```
    @BeforeEach
```

```
    public void setUp() {
```

```

wallet = new Wallet("Wallet123", 1000.0);

user = new User("U001", "Alice", "alice@example.com", "0xWalletAddress123", wallet);

transactionService = new TransactionService();
}

```

```

@Test

public void testEndToEndAssetTransfer() {

    user.initiateSendAsset("BTC", "0xRecipientAddress", 0.5);


    // Assert wallet balance updated

    assertEquals(999.5, wallet.getBalance(), "Balance should reflect asset transfer");


    // Assert transaction is processed and recorded

    assertEquals(1, wallet.getTransactionHistory().size(), "Transaction should be recorded
in transaction history");

    assertEquals("Completed", wallet.getTransactionHistory().get(0).getStatus(),
"Transaction status should be completed");

}

}

` ``

```

System Test: End-to-End Smart Contract Execution

```

` `` java

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

```



```

public class SmartContractSystemTest {

    private User user;

    private Wallet wallet;

    @BeforeEach

    public void setUp() {

        wallet = new Wallet("Wallet123", 1000.0);

        user = new User("U001", "Alice", "alice@example.com", "0xWalletAddress123", wallet);

    }

    @Test

    public void testEndToEndSmartContractExecution() {

        user.initiateSmartContract("SC001", "{parameters}");

        // Assert smart contract is processed and recorded

        assertEquals(1, wallet.getTransactionHistory().size(), "Smart contract execution should
be recorded");

        assertEquals("Completed", wallet.getTransactionHistory().get(0).getStatus(), "Smart
contract status should be completed");

    }

}

...

#### **System Test: End-to-End Social Recovery**

```java

```

```

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class SocialRecoverySystemTest {

 private User user;

 private SecuritySettings securitySettings;

 @BeforeEach
 public void setUp() {
 user = new User("U001", "Alice", "alice@example.com", "0xWalletAddress123");
 securitySettings = new SecuritySettings();
 }

 @Test
 public void testEndToEndSocialRecovery() {
 boolean result = securitySettings.initiateSocialRecovery(user);
 assertTrue(result, "Social recovery should be completed successfully for the user");
 }
}

```

---

### Summary of Tests

- **Unit Tests**: Test individual methods in classes like `User`, `Wallet`, and `Transaction`.
- **Integration Tests**: Test interactions between `User` and `Wallet`, especially for operations like asset transfer and smart contract execution.
- **System Tests**: Test the complete workflow for asset transfer, smart contract execution, and social recovery from start to end, ensuring the system functions as expected.

These tests provide a comprehensive testing suite to verify the system's functionality and ensure that each component and interaction works correctly.