

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.

AI Model

I'll help analyze the system description to identify both functional and non-functional requirements for a smart wallet system.

```
# Smart Wallet System Requirements Analysis
```

```
## Functional Requirements
```

```
### Asset Management
```

1. Support storage and transfer of multiple cryptocurrencies
2. Handle various digital assets (tokens, NFTs)
3. Execute cryptocurrency transfers between accounts
4. Display balances for all stored digital assets
5. Support batch transactions (multiple operations in one transaction)

Smart Contract Integration

1. Execute smart contract operations
2. Interact with decentralized applications (dApps)
3. Support programmable transactions
4. Enable automated transaction execution
5. Implement account abstraction features

Security Operations

1. Implement multi-signature authentication
2. Provide social recovery mechanisms
3. Enable setting and enforcement of spending limits
4. Verify transaction authenticity
5. Support key management functions

Transaction Management

1. Process cryptocurrency payments
2. Optimize gas fees for transactions
3. Support transaction scheduling
4. Enable transaction batching
5. Provide transaction history and status tracking

Interface Operations

1. Present unified interface for all supported assets
2. Display real-time asset values
3. Show transaction status updates
4. Provide dApp interaction interface

5. Enable customization of security settings

Non-Functional Requirements

Security

1. Secure storage of private keys
2. Encrypted communication channels
3. Secure authentication mechanisms
4. Protection against unauthorized access
5. Regular security audits compliance

Performance

1. Fast transaction processing
2. Quick response time for user interactions
3. Efficient gas fee optimization
4. Minimal latency in dApp interactions
5. Scalable transaction handling

Usability

1. User-friendly interface
2. Intuitive navigation
3. Clear transaction information display
4. Easy-to-understand security settings
5. Simplified dApp interaction process

Reliability

1. High system availability
2. Consistent transaction execution
3. Accurate balance tracking
4. Reliable smart contract interaction
5. Dependable recovery mechanisms

Compatibility

1. Cross-blockchain support
2. Multiple cryptocurrency compatibility
3. Various dApp integration capability
4. Different device support
5. Multiple operating system compatibility

Maintainability

1. Upgradeable smart contracts
2. Modular system architecture
3. Clear system documentation
4. Version control management
5. Easy system updates

Quality Attributes

Interoperability

- Seamless interaction with various blockchain networks
- Compatible with existing Web3 infrastructure
- Integration with traditional payment systems

Scalability

- Handle increasing number of users
- Support growing transaction volumes
- Manage multiple asset types efficiently

Compliance

- Adherence to regulatory requirements
- Compliance with blockchain standards
- Implementation of required security protocols

Recovery

- Robust backup mechanisms
- Clear recovery procedures
- Multiple recovery options

Accessibility

- Support for multiple languages
- Accommodation of different user skill levels
- Clear error messages and guidance

The requirements analysis above categorizes the essential functional and non-functional aspects of a smart wallet system based on the provided description. Key points to note:

1. Functional requirements focus on specific features and capabilities the system must provide, like asset management, smart contract interactions, and security operations.

2. Non-functional requirements address the quality attributes and constraints that affect system behavior, such as security, performance, and usability.
3. Quality attributes provide additional cross-cutting concerns that affect multiple aspects of the system.

Would you like me to elaborate on any particular category of requirements or discuss specific implementation considerations for any of these features?

User

Create a use case model for the system.

AI Model

I'll create a use case model for the smart wallet system, including a diagram and detailed use case descriptions.

stateDiagram-v2

```
state "Smart Wallet System" as SWS {  
    state "Asset Management" as AM {  
        [*] --> ViewBalance  
        [*] --> TransferAssets  
        [*] --> ManageTokens  
        [*] --> TrackNFTs  
    }  
}
```

```
state "Security Management" as SM {  
    [*] --> SetupMultiSig  
    [*] --> ConfigureRecovery  
    [*] --> ManageSpendingLimits  
    [*] --> AuthenticateUser
```

}

state "Transaction Management" as TM {

[*] --> ExecuteTransaction

[*] --> BatchTransactions

[*] --> OptimizeGasFees

[*] --> ScheduleTransactions

}

state "DApp Integration" as DI {

[*] --> ConnectToDApp

[*] --> ExecuteSmartContract

[*] --> AutomateTransactions

}

}

Smart Wallet System Use Cases

Primary Actors

1. Wallet Owner
2. DApp Provider
3. Smart Contract
4. Network Validator
5. Recovery Contact

Asset Management Use Cases

1. View Balance

- ****Actor****: Wallet Owner
- ****Description****: User checks balances of all digital assets
- ****Preconditions****: Wallet is authenticated
- ****Main Flow****:
 1. User opens wallet interface
 2. System retrieves current balances
 3. System displays balances for all assets
- ****Alternative Flows****:
 - Network connection error
 - Asset data unavailable
- ****Postconditions****: Updated balances displayed

2. Transfer Assets

- ****Actor****: Wallet Owner
- ****Description****: User transfers digital assets to another address
- ****Preconditions****:
 - Wallet is authenticated
 - Sufficient balance available
- ****Main Flow****:
 1. User initiates transfer
 2. System validates recipient address
 3. User confirms transaction details
 4. System executes transfer
- ****Alternative Flows****:
 - Insufficient balance

- Invalid address
- Network congestion
- **Postconditions**: Assets transferred, balances updated

Security Management Use Cases

1. Setup Multi-Signature

- **Actor**: Wallet Owner
- **Description**: Configure multi-signature requirements
- **Preconditions**: Wallet created
- **Main Flow**:
 1. User selects multi-sig setup
 2. Adds required signers
 3. Sets approval threshold
 4. Confirms configuration
- **Alternative Flows**:
 - Invalid signer addresses
 - Threshold configuration error
- **Postconditions**: Multi-sig activated

2. Configure Recovery

- **Actors**: Wallet Owner, Recovery Contact
- **Description**: Set up wallet recovery options
- **Main Flow**:
 1. User selects recovery setup
 2. Adds recovery contacts

3. Sets recovery parameters

4. System stores recovery data

- **Alternative Flows**:

- Invalid recovery contact

- Recovery setup failure

Transaction Management Use Cases

1. Execute Transaction

- **Actor**: Wallet Owner

- **Description**: Process a blockchain transaction

- **Preconditions**:

- Wallet authenticated

- Sufficient funds/gas

- **Main Flow**:

1. User initiates transaction

2. System estimates gas fees

3. User confirms transaction

4. System broadcasts to network

- **Alternative Flows**:

- Insufficient gas

- Network failure

- Transaction rejected

2. Batch Transactions

- **Actor**: Wallet Owner

- **Description**: Execute multiple transactions as one

- **Main Flow**:

1. User adds multiple transactions
2. System calculates total gas
3. User confirms batch
4. System executes all transactions

- **Alternative Flows**:

- Partial batch failure
- Gas estimation error

DApp Integration Use Cases

1. Connect to DApp

- **Actors**: Wallet Owner, DApp Provider

- **Description**: Connect wallet to decentralized application

- **Main Flow**:

1. User initiates DApp connection
2. DApp requests permissions
3. User approves connection
4. System establishes connection

- **Alternative Flows**:

- Connection rejection
- Permission denial

2. Execute Smart Contract

- **Actors**: Wallet Owner, Smart Contract

- **Description**: Interact with smart contract functions

- **Main Flow**:

1. User selects contract function
2. System loads contract interface
3. User inputs parameters
4. System executes contract call

- **Alternative Flows**:

- Contract execution failure
- Invalid parameters

Special Requirements

1. **Performance**:

- Transaction confirmation < 30 seconds
- UI response time < 2 seconds

2. **Security**:

- Encrypted data storage
- Secure key management
- Multi-factor authentication

3. **Availability**:

- 99.9% uptime
- Redundant connections

4. **Usability**:

- Intuitive interface

- Clear error messages
- Guided recovery process

Technical Constraints

1. Blockchain network compatibility
2. Gas fee limitations
3. Network congestion handling
4. Smart contract limitations
5. Cross-chain integration requirements

Business Rules

1. Transaction limits enforcement
2. Regulatory compliance
3. Fee structure adherence
4. Recovery process validation
5. Multi-signature policies

I'll create a use case model for the smart wallet system, including a diagram and detailed use case descriptions.

```mermaid

stateDiagram-v2

```
state "Smart Wallet System" as SWS {
 state "Asset Management" as AM {
```

```
[*] --> ViewBalance
[*] --> TransferAssets
[*] --> ManageTokens
[*] --> TrackNFTs
}
```

```
state "Security Management" as SM {
 [*] --> SetupMultiSig
 [*] --> ConfigureRecovery
 [*] --> ManageSpendingLimits
 [*] --> AuthenticateUser
}
```

```
state "Transaction Management" as TM {
 [*] --> ExecuteTransaction
 [*] --> BatchTransactions
 [*] --> OptimizeGasFees
 [*] --> ScheduleTransactions
}
```

```
state "DApp Integration" as DI {
 [*] --> ConnectToDApp
 [*] --> ExecuteSmartContract
 [*] --> AutomateTransactions
}
}
```

...

## # Smart Wallet System Use Cases

### ## Primary Actors

1. Wallet Owner
2. DApp Provider
3. Smart Contract
4. Network Validator
5. Recovery Contact

### ## Asset Management Use Cases

#### ### 1. View Balance

- **Actor**: Wallet Owner
- **Description**: User checks balances of all digital assets
- **Preconditions**: Wallet is authenticated
- **Main Flow**:
  1. User opens wallet interface
  2. System retrieves current balances
  3. System displays balances for all assets
- **Alternative Flows**:
  - Network connection error

- Asset data unavailable
- **Postconditions**: Updated balances displayed

### ### 2. Transfer Assets

- **Actor**: Wallet Owner
- **Description**: User transfers digital assets to another address
- **Preconditions**:
  - Wallet is authenticated
  - Sufficient balance available
- **Main Flow**:
  1. User initiates transfer
  2. System validates recipient address
  3. User confirms transaction details
  4. System executes transfer
- **Alternative Flows**:
  - Insufficient balance
  - Invalid address
  - Network congestion
- **Postconditions**: Assets transferred, balances updated

## ## Security Management Use Cases

### ### 1. Setup Multi-Signature

- **Actor**: Wallet Owner
- **Description**: Configure multi-signature requirements
- **Preconditions**: Wallet created



- **Main Flow**:

1. User selects multi-sig setup
2. Adds required signers
3. Sets approval threshold
4. Confirms configuration

- **Alternative Flows**:

- Invalid signer addresses
- Threshold configuration error

- **Postconditions**: Multi-sig activated

### ### 2. Configure Recovery

- **Actors**: Wallet Owner, Recovery Contact

- **Description**: Set up wallet recovery options

- **Main Flow**:

1. User selects recovery setup
2. Adds recovery contacts
3. Sets recovery parameters
4. System stores recovery data

- **Alternative Flows**:

- Invalid recovery contact
- Recovery setup failure

## ## Transaction Management Use Cases

### ### 1. Execute Transaction

- **Actor**: Wallet Owner

- **Description**: Process a blockchain transaction

- **Preconditions**:

- Wallet authenticated

- Sufficient funds/gas

- **Main Flow**:

1. User initiates transaction

2. System estimates gas fees

3. User confirms transaction

4. System broadcasts to network

- **Alternative Flows**:

- Insufficient gas

- Network failure

- Transaction rejected

## ### 2. Batch Transactions

- **Actor**: Wallet Owner

- **Description**: Execute multiple transactions as one

- **Main Flow**:

1. User adds multiple transactions

2. System calculates total gas

3. User confirms batch

4. System executes all transactions

- **Alternative Flows**:

- Partial batch failure

- Gas estimation error

## ## DApp Integration Use Cases

### ### 1. Connect to DApp

- **Actors**: Wallet Owner, DApp Provider
- **Description**: Connect wallet to decentralized application
- **Main Flow**:
  1. User initiates DApp connection
  2. DApp requests permissions
  3. User approves connection
  4. System establishes connection
- **Alternative Flows**:
  - Connection rejection
  - Permission denial

### ### 2. Execute Smart Contract

- **Actors**: Wallet Owner, Smart Contract
- **Description**: Interact with smart contract functions
- **Main Flow**:
  1. User selects contract function
  2. System loads contract interface
  3. User inputs parameters
  4. System executes contract call
- **Alternative Flows**:
  - Contract execution failure
  - Invalid parameters

## ## Special Requirements

### 1. **Performance**:

- Transaction confirmation < 30 seconds
- UI response time < 2 seconds

### 2. **Security**:

- Encrypted data storage
- Secure key management
- Multi-factor authentication

### 3. **Availability**:

- 99.9% uptime
- Redundant connections

### 4. **Usability**:

- Intuitive interface
- Clear error messages
- Guided recovery process

## ## Technical Constraints

1. Blockchain network compatibility
2. Gas fee limitations
3. Network congestion handling
4. Smart contract limitations
5. Cross-chain integration requirements

## ## Business Rules

1. Transaction limits enforcement
2. Regulatory compliance
3. Fee structure adherence
4. Recovery process validation
5. Multi-signature policies

This use case model provides a comprehensive view of the smart wallet system's functionality from a user perspective. The model is organized into four main categories:

1. Asset Management
2. Security Management
3. Transaction Management
4. DApp Integration

Each category contains detailed use cases with actors, flows, and conditions. The model also includes special requirements, technical constraints, and business rules that affect the use cases.

Key points about the model:

1. It shows both basic operations (like viewing balances) and advanced features (like multi-signature setup)
2. It identifies multiple actors and their interactions
3. It includes alternative flows for error conditions
4. It specifies preconditions and postconditions for major operations

Would you like me to elaborate on any particular use case or add additional scenarios to the model?

## **User**

Create use case specifications for identified use cases.

## **AI Model**

I'll create detailed use case specifications for the key use cases identified in the smart wallet system.

### # Smart Wallet System Use Case Specifications

#### ## 1. Asset Management

##### ### UC-1: View Balance

##### #### Description

Allows users to view their current balance for all digital assets in their wallet.

##### #### Primary Actor

- Wallet Owner

##### #### Stakeholders and Interests

- Wallet Owner: Wants accurate, real-time balance information
- System Provider: Must ensure accurate data display
- Network Validators: Must provide current blockchain state

##### #### Preconditions

1. User has authenticated into the wallet

2. Wallet has active network connection
3. User has at least one asset in wallet

#### #### Postconditions

1. Current balances are displayed
2. Transaction history is updated
3. Asset values are calculated in selected currency

#### #### Main Success Scenario

1. User opens wallet application
2. System authenticates user credentials
3. System retrieves current blockchain state
4. System calculates current balance for each asset
5. System converts values to selected display currency
6. System displays:
  - Individual asset balances
  - Total portfolio value
  - Change in value (24h)
  - Asset distribution

#### #### Extensions/Alternative Flows

- 1a. Authentication fails
  1. System displays authentication error
  2. System prompts for re-authentication
  3. User re-enters credentials

### 3a. Network connection fails

1. System displays network error
2. System attempts to reconnect
3. System shows last known balances with timestamp

### 4a. Asset price data unavailable

1. System displays "Price Unavailable"
2. Shows balance in native token only
3. Marks total portfolio value as incomplete

### #### Special Requirements

- Response time < 2 seconds
- 99.9% accuracy in balance reporting
- Real-time price updates
- Support for multiple currencies

### #### Technology and Data Variations

1. Support for multiple blockchain networks
2. Various token standards (ERC-20, ERC-721, etc.)
3. Multiple price feed sources

### #### Frequency of Occurrence

- High (multiple times per day per user)

### ### UC-2: Transfer Assets

#### #### Description



Enables users to send digital assets to other blockchain addresses.

#### #### Primary Actor

- Wallet Owner

#### #### Stakeholders and Interests

- Wallet Owner: Wants secure, confirmed transfers
- Recipient: Wants to receive correct amount
- Network Validators: Need to process transaction
- System Provider: Must ensure secure transaction handling

#### #### Preconditions

1. User is authenticated
2. Sufficient balance available
3. Network is operational
4. Gas fees can be estimated

#### #### Postconditions

1. Transaction is broadcast to network
2. Balance is updated
3. Transaction is recorded in history
4. Recipient receives assets

#### #### Main Success Scenario

1. User selects "Transfer" function
2. User inputs or selects recipient address

3. User enters transfer amount
4. System validates:
  - Address format
  - Sufficient balance
  - Network status
5. System estimates gas fees
6. User confirms transaction details
7. System signs transaction
8. System broadcasts to network
9. System monitors confirmation
10. System updates UI with status

#### #### Extensions/Alternative Flows

- 2a. Invalid address format
  1. System highlights error
  2. Prompts for correction
  3. Provides address format guide
- 4a. Insufficient balance
  1. System displays insufficient funds error
  2. Shows maximum available amount
  3. Prompts for amount adjustment
- 5a. Gas estimation fails
  1. System displays gas estimation error
  2. Offers manual gas setting

### 3. Provides retry option

#### #### Special Requirements

- Transaction confirmation notification
- Gas fee optimization
- Address book integration
- QR code scanner support

#### #### Technology and Data Variations

1. Multiple token standards support
2. Cross-chain transfer capabilities
3. Various gas fee structures

#### #### Frequency of Occurrence

- Medium (several times per week per user)

## ## 2. Security Management

### ### UC-3: Setup Multi-Signature

#### #### Description

Configures multi-signature requirements for wallet operations.

#### #### Primary Actor

- Wallet Owner

#### #### Stakeholders and Interests

- Wallet Owner: Wants secure multi-sig setup
- Co-Signers: Need clear signing responsibilities
- System Provider: Must ensure secure setup

#### #### Preconditions

1. Wallet is initialized
2. User has required co-signer addresses
3. Smart contract deployment possible

#### #### Postconditions

1. Multi-sig contract deployed
2. Co-signers configured
3. Threshold set
4. Old access revoked

#### #### Main Success Scenario

1. User selects multi-sig setup
2. System explains multi-sig concepts
3. User adds co-signer addresses
4. User sets approval threshold
5. System validates all addresses
6. System generates multi-sig contract
7. User approves gas fees
8. System deploys contract
9. System configures wallet with new contract
10. System sends notifications to co-signers

#### #### Extensions/Alternative Flows

##### 3a. Invalid co-signer address

1. System highlights invalid address
2. Provides address format guide
3. Requests correction

##### 4a. Invalid threshold configuration

1. System shows allowed threshold range
2. Explains threshold implications
3. Prompts for adjustment

##### 8a. Contract deployment fails

1. System shows deployment error
2. Offers retry option
3. Provides manual deployment option

#### #### Special Requirements

- Clear security implications
- Co-signer verification
- Backup access methods
- Gas fee estimation

#### #### Technology and Data Variations

1. Different multi-sig contract standards
2. Various blockchain network support

### 3. Multiple approval schemes

#### #### Frequency of Occurrence

- Low (once per wallet setup, occasional updates)

[Additional use cases would continue in this format...]

#### ## Additional Considerations

##### ### Security Requirements

1. Encryption of local data
2. Secure key storage
3. Network security protocols
4. Authentication mechanisms
5. Recovery procedures

##### ### Performance Requirements

1. Transaction processing time
2. UI response time
3. Network operation timeouts
4. Recovery time objectives
5. Concurrent operation handling

##### ### Usability Requirements

1. Interface accessibility
2. Error message clarity

3. Operation confirmation
4. Help documentation
5. User guidance

### ### Business Rules

1. Transaction limits
2. Fee structures
3. Recovery policies
4. Multi-signature policies
5. Regulatory compliance requirements

This specification document provides detailed descriptions for key use cases in the smart wallet system. Each use case specification includes:

1. Clear description and scope
2. Primary actors and stakeholders
3. Pre and postconditions
4. Main success scenario
5. Alternative flows
6. Special requirements
7. Technical variations
8. Frequency of occurrence

I've provided detailed specifications for three critical use cases:

1. View Balance (basic but frequently used)

2. Transfer Assets (core functionality)
3. Setup Multi-Signature (complex security feature)

Would you like me to:

1. Provide specifications for additional use cases?
2. Add more detail to any particular section?
3. Focus on specific aspects like security or performance requirements?

## User

Create a domain model based on use case specifications.

## AI Model

I'll create a domain model for the smart wallet system that captures the essential concepts, their attributes, and relationships based on the use case specifications.

classDiagram

Wallet --|> SmartContract

Wallet "1" \*-- "many" Asset

Wallet "1" \*-- "many" Transaction

Wallet "1" \*-- "1" SecuritySettings

Wallet "1" \*-- "many" DAppConnection

Transaction "1" \*-- "1" GasFee

Transaction "1" \*-- "1" TransactionStatus

SecuritySettings "1" \*-- "1" MultiSigConfig

SecuritySettings "1" \*-- "1" RecoveryConfig



SecuritySettings "1" \*-- "many" SpendingLimit

WalletOwner "1" -- "many" Wallet

WalletOwner "1" -- "many" RecoveryContact

Asset "1" -- "many" Transaction

DApp "1" -- "many" SmartContract

DApp "1" -- "many" DAppConnection

```
class Wallet {
 +walletAddress: String
 +networkType: String
 +creationDate: DateTime
 +balance: Decimal
 +status: WalletStatus
 +createTransaction()
 +connectToDApp()
 +updateSecuritySettings()
 +calculateTotalBalance()
}
```

```
class Asset {
 +assetId: String
 +type: AssetType
 +amount: Decimal
```

```
+symbol: String
+decimals: Integer
+lastUpdated: DateTime
+getCurrentPrice()
+calculateValue()
}
```

```
class Transaction {
 +transactionId: String
 +fromAddress: String
 +toAddress: String
 +amount: Decimal
 +timestamp: DateTime
 +nonce: Integer
 +data: String
 +sign()
 +broadcast()
 +confirm()
}
```

```
class SecuritySettings {
 +securityLevel: Level
 +lastUpdated: DateTime
 +twoFactorEnabled: Boolean
 +updateMultiSig()
 +configureRecovery()
```

```
+setSpendingLimits()
}
```

```
class MultiSigConfig {
 +requiredSignatures: Integer
 +signers: List~Address~
 +threshold: Integer
 +timelock: Duration
 +addSigner()
 +removeSigner()
 +updateThreshold()
}
```

```
class RecoveryConfig {
 +recoveryMethod: RecoveryType
 +guardians: List~Address~
 +cooldownPeriod: Duration
 +addGuardian()
 +removeGuardian()
 +initiateRecovery()
}
```

```
class SpendingLimit {
 +limitType: LimitType
 +amount: Decimal
 +period: Duration
```

```
+assetType: AssetType
+checkLimit()
+updateLimit()
}
```

```
class WalletOwner {
 +userId: String
 +publicKey: String
 +email: String
 +lastAccess: DateTime
 +createWallet()
 +manageAssets()
 +configureSettings()
}
```

```
class DAppConnection {
 +connectionId: String
 +permissions: List~Permission~
 +status: ConnectionStatus
 +lastActive: DateTime
 +connect()
 +disconnect()
 +updatePermissions()
}
```

```
class SmartContract {
```

```
+contractAddress: String
+abi: String
+bytecode: String
+deploy()
+execute()
+validate()
}
```

```
class GasFee {
 +baseGas: Decimal
 +maxGas: Decimal
 +priorityFee: Decimal
 +estimateGas()
 +optimizeFee()
}
```

```
class TransactionStatus {
 +status: Status
 +confirmations: Integer
 +blockNumber: Integer
 +timestamp: DateTime
 +checkStatus()
 +updateStatus()
}
```

# Smart Wallet Domain Model Specifications

## ## Core Entities

### ### Wallet

The central entity representing a smart wallet instance.

- **\*\*Attributes\*\***:
  - walletAddress: Unique blockchain address
  - networkType: Blockchain network identifier
  - creationDate: Wallet creation timestamp
  - balance: Current total balance
  - status: Active/Inactive/Locked
- **\*\*Responsibilities\*\***:
  - Manage assets and transactions
  - Handle security settings
  - Connect to DApps
  - Execute smart contracts

### ### Asset

Represents any digital asset stored in the wallet.

- **\*\*Attributes\*\***:
  - assetId: Unique identifier
  - type: Cryptocurrency/Token/NFT
  - amount: Quantity held
  - symbol: Asset symbol
  - decimals: Decimal places for amount
- **\*\*Relationships\*\***:

- Belongs to one Wallet
- Involved in many Transactions

### ### Transaction

Records all blockchain operations.

- **\*\*Attributes\*\***:
  - transactionId: Unique transaction hash
  - fromAddress: Sender address
  - toAddress: Recipient address
  - amount: Transaction amount
  - timestamp: Operation time
  - nonce: Transaction sequence number
- **\*\*Relationships\*\***:
  - Associated with one Asset
  - Has one GasFee
  - Has one TransactionStatus

## ## Security Components

### ### SecuritySettings

Manages wallet security configuration.

- **\*\*Attributes\*\***:
  - securityLevel: High/Medium/Low
  - lastUpdated: Last modification time
  - twoFactorEnabled: 2FA status
- **\*\*Components\*\***:

- MultiSigConfig
- RecoveryConfig
- SpendingLimit

### ### MultiSigConfig

Controls multi-signature requirements.

- **\*\*Attributes\*\***:
  - requiredSignatures: Number of required signatures
  - signers: List of authorized signers
  - threshold: Approval threshold
  - timelock: Waiting period
- **\*\*Relationships\*\***:
  - Part of SecuritySettings

### ### RecoveryConfig

Manages wallet recovery options.

- **\*\*Attributes\*\***:
  - recoveryMethod: Social/Hardware/Seed
  - guardians: Recovery contacts
  - cooldownPeriod: Recovery waiting period
- **\*\*Relationships\*\***:
  - Part of SecuritySettings

## ## DApp Integration

### ### DAppConnection



Manages connections to decentralized applications.

- **Attributes**:

- `connectionId`: Unique connection identifier
- `permissions`: Granted permissions
- `status`: Connected/Disconnected
- `lastActive`: Last activity timestamp

- **Relationships**:

- Belongs to one Wallet
- Connected to one DApp

### SmartContract

Represents deployable/executable smart contracts.

- **Attributes**:

- `contractAddress`: Contract address
- `abi`: Contract interface
- `bytecode`: Compiled contract code

- **Relationships**:

- Associated with DApp
- Inherited by Wallet

## Transaction Components

### GasFee

Handles transaction fee calculations.

- **Attributes**:

- `baseGas`: Base gas price

- maxGas: Maximum gas allowance
- priorityFee: Priority fee component
- **Relationships**:
  - Belongs to one Transaction

### TransactionStatus

Tracks transaction state.

- **Attributes**:
  - status: Pending/Confirmed/Failed
  - confirmations: Number of confirmations
  - blockNumber: Confirmation block
  - timestamp: Status update time
- **Relationships**:
  - Belongs to one Transaction

## Enumerations

### AssetType

- Cryptocurrency
- Token
- NFT

### RecoveryType

- Social
- Hardware
- Seed

### ### LimitType

- Daily
- Weekly
- Monthly
- PerTransaction

### ### Status

- Pending
- Confirmed
- Failed
- Rejected

### ### ConnectionStatus

- Connected
- Disconnected
- Pending
- Rejected

## ## Value Objects

### ### Permission

- Read
- Write
- Execute
- Full

### Level

- High
- Medium
- Low

The domain model above captures the key concepts and relationships in the smart wallet system. Key aspects include:

**1. Core Functionality:**

- Wallet management
- Asset tracking
- Transaction processing
- Security controls

**2. Important Relationships:**

- Wallet to Assets (composition)
- Wallet to Transactions (composition)
- Wallet to SecuritySettings (composition)
- WalletOwner to Wallet (association)

**3. Security Features:**

- Multi-signature support
- Recovery mechanisms
- Spending limits
- DApp permissions

Would you like me to:

1. Elaborate on any specific entity or relationship?
2. Add additional attributes or methods?

3. Explain the reasoning behind any particular design decision?

## User

Identify system operations from use case specifications.

## AI Model

I'll identify and specify the system operations based on the use case specifications, including their contracts (pre/postconditions).

# Smart Wallet System Operations

## 1. Asset Management Operations

### 1.1 getWalletBalance(walletAddress: String): Map<AssetID, Amount>

**Description**: Retrieves current balance for all assets in wallet

**Contract**:

...

preconditions:

- walletAddress is valid blockchain address
- wallet exists in system
- network connection is available

postconditions:

- returns map of asset IDs to current balances
- all balances are up-to-date with blockchain state
- includes zero balances for owned assets

exceptions:

- InvalidAddressException if address format invalid

- NetworkException if connection fails
- WalletNotFoundException if wallet doesn't exist

\ \ \

### 1.2 transferAssets(from: Address, to: Address, asset: AssetID, amount: Decimal):  
TransactionID

**\*\*Description\*\***: Initiates asset transfer between addresses

**\*\*Contract\*\***:

\ \ \

preconditions:

- sender address is valid and authenticated
- recipient address is valid
- asset exists in wallet
- amount > 0
- balance >= amount + estimated gas fee

postconditions:

- transaction created and broadcast to network
- pending transaction recorded in history
- available balance updated
- gas fee reserved

exceptions:

- InsufficientBalanceException
- InvalidAddressException
- InvalidAmountException
- NetworkException

\ \ \

### ### 1.3 estimateGasFee(transaction: TransactionDetails): GasFeeEstimate

**\*\*Description\*\***: Calculates estimated gas fee for transaction

**\*\*Contract\*\***:

\ \ \

preconditions:

- transaction details are complete
- network connection available
- gas oracle accessible

postconditions:

- returns estimated gas fee range
- includes base fee and priority fee
- provides low/medium/high options

exceptions:

- NetworkException
- GasEstimationFailedException

\ \ \

## ## 2. Security Management Operations

### ### 2.1 setupMultiSig(walletAddress: String, signers: List<Address>, threshold: Int): MultiSigConfig

**\*\*Description\*\***: Configures multi-signature requirements for wallet

**\*\*Contract\*\***:

\ \ \

preconditions:

- wallet owner is authenticated
- all signer addresses are valid
- threshold  $\leq$  number of signers
- threshold  $> 0$

postconditions:

- multi-sig contract deployed
- wallet configured with multi-sig
- signers notified of configuration
- old security settings archived

exceptions:

- InvalidThresholdException
- InvalidSignerException
- DeploymentFailedException

...

### 2.2 configureRecovery(walletAddress: String, method: RecoveryMethod, guardians: List<Address>): RecoveryConfig

**Description**: Sets up wallet recovery mechanisms

**Contract**:

...

preconditions:

- wallet owner is authenticated
- recovery method is valid
- guardian addresses are valid
- minimum required guardians provided

postconditions:



- recovery configuration stored
- guardians notified and confirmed
- recovery contract deployed if required
- backup procedures documented

exceptions:

- InvalidGuardianException
- InvalidMethodException
- ConfigurationFailedException

\ \ \

### 2.3 setSpendingLimits(walletAddress: String, limits: List<SpendingLimit>): Boolean

**\*\*Description\*\***: Configures spending limits for wallet

**\*\*Contract\*\***:

\ \ \

preconditions:

- wallet owner is authenticated
- limits are within allowed ranges
- asset types are valid

postconditions:

- spending limits activated
- notification sent to owner
- limit monitoring enabled

exceptions:

- InvalidLimitException
- ConfigurationFailedException

\ \ \

## ## 3. Transaction Management Operations

### ### 3.1 createTransaction(transactionDetails: TransactionDetails): Transaction

**\*\*Description\*\***: Creates new transaction based on provided details

**\*\*Contract\*\***:

` ``

preconditions:

- sender authenticated
- valid transaction details
- sufficient balance available
- within spending limits

postconditions:

- transaction object created
- gas fee estimated
- nonce assigned
- ready for signing

exceptions:

- InvalidDetailsException
- InsufficientBalanceException
- SpendingLimitException

` ``

### ### 3.2 signTransaction(transaction: Transaction, signer: Address): SignedTransaction

**\*\*Description\*\***: Signs transaction with private key

**\*\*Contract\*\***:

\ \ \

preconditions:

- transaction is valid
- signer is authorized
- private key available

postconditions:

- transaction signed
- signature verified
- multi-sig status updated if applicable

exceptions:

- SigningFailedException
- UnauthorizedSignerException

\ \ \

### 3.3 broadcastTransaction(signedTransaction: SignedTransaction): TransactionHash

**\*\*Description\*\***: Broadcasts signed transaction to network

**\*\*Contract\*\***:

\ \ \

preconditions:

- transaction fully signed
- network connection available
- gas fee confirmed

postconditions:

- transaction broadcast to network
- transaction hash returned
- pending status recorded

exceptions:

- NetworkException
- BroadcastFailedException

\ \ \

## ## 4. DApp Integration Operations

### 4.1 connectToDApp(walletAddress: String, dappUrl: String, permissions:  
List<Permission>): Connection

**\*\*Description\*\***: Establishes connection with decentralized application

**\*\*Contract\*\***:

\ \ \

preconditions:

- wallet authenticated
- DApp URL valid
- permissions specified

postconditions:

- connection established
- permissions granted
- connection ID returned

exceptions:

- ConnectionFailedException
- InvalidPermissionException

\ \ \

### 4.2 executeSmartContract(contract: Address, method: String, params:  
List<Parameter>): Result

**\*\*Description\*\***: Executes smart contract method

**\*\*Contract\*\***:

\ \ \

preconditions:

- contract address valid
- method exists in ABI
- parameters match method signature
- caller has execution permission

postconditions:

- method executed
- result returned
- gas consumed
- state updated

exceptions:

- ExecutionFailedException
- InvalidParameterException
- OutOfGasException

\ \ \

## ## 5. Utility Operations

### 5.1 validateAddress(address: String, network: String): Boolean

**\*\*Description\*\***: Validates blockchain address format

**\*\*Contract\*\***:

\ \ \

preconditions:

- address string provided
- network type specified

postconditions:

- returns validation result
- validation reason provided if false

exceptions:

- InvalidNetworkException

`, ``,

### 5.2 encryptData(data: String, key: EncryptionKey): EncryptedData

**Description**: Encrypts sensitive wallet data

**Contract**:

`, ``,

preconditions:

- valid data provided
- encryption key available
- data within size limits

postconditions:

- data encrypted
- encryption verified
- original data securely deleted

exceptions:

- EncryptionFailedException
- InvalidKeyException

`, ``,

### 5.3 monitorTransactionStatus(transactionHash: String): TransactionStatus

**\*\*Description\*\***: Monitors and updates transaction status

**\*\*Contract\*\***:

\ \ \

preconditions:

- valid transaction hash
- network connection available

postconditions:

- current status returned
- status history updated
- notifications sent if configured

exceptions:

- InvalidHashException
- NetworkException

\ \ \

## Common Patterns

### Operation Categories:

1. **\*\*Query Operations\*\***

- Don't modify system state
- Return requested information
- Example: getWalletBalance()

2. **\*\*Command Operations\*\***

- Modify system state

- Return success/failure
- Example: transferAssets()

### 3. **\*\*Validation Operations\*\***

- Verify data/conditions
- Return boolean result
- Example: validateAddress()

## ### Error Handling:

### 1. **\*\*Validation Errors\*\***

- Input validation
- Permission checking
- Format verification

### 2. **\*\*System Errors\*\***

- Network issues
- Resource unavailable
- Internal errors

### 3. **\*\*Business Rule Violations\*\***

- Spending limits
- Multi-sig requirements
- Time constraints

The system operations identified above cover:

1. Core wallet functionality:



- Asset management
  - Transaction processing
  - Security controls
  - DApp integration
2. Each operation includes:
    - Clear description
    - Preconditions
    - Postconditions
    - Exception cases
  3. Operations are grouped by functionality area and follow common patterns for consistency

Would you like me to:

1. Add more detail to any specific operation?
2. Include additional operations for specific use cases?
3. Elaborate on the error handling or validation rules?

## **User**

Create design sequence diagrams for system operations.

## **AI Model**

I'll create sequence diagrams for key system operations, showing the interaction between objects and components.

sequenceDiagram

actor User

participant WI as WalletInterface

participant TC as TransactionController

participant WM as WalletManager

participant VM as ValidationManager

participant GE as GasEstimator

participant BC as BlockchainConnector

User->>WI: initiateTransfer(to, asset, amount)

activate WI

WI->>TC: createTransaction(from, to, asset, amount)

activate TC

TC->>VM: validateTransaction(from, to, asset, amount)

activate VM

VM-->>TC: validationResult

deactivate VM

TC->>WM: checkBalance(from, asset)

activate WM

WM-->>TC: balanceResult

deactivate WM

TC->>GE: estimateGasFee(txDetails)

activate GE

GE-->>TC: gasFeeEstimate

deactivate GE

TC-->>WI: transactionDetails

deactivate TC

WI->>User: showConfirmation(transactionDetails)

User->>WI: confirmTransaction()

WI->>TC: executeTransaction(signedTx)

activate TC

TC->>BC: broadcastTransaction(signedTx)

activate BC

BC-->>TC: transactionHash

deactivate BC

TC-->>WI: transactionStatus

deactivate TC

WI-->>User: displayResult()

deactivate WI

sequenceDiagram

actor Owner

participant WI as WalletInterface

participant SC as SecurityController

participant CM as ContractManager

participant NM as NotificationManager

participant BC as BlockchainConnector

Owner->>WI: setupMultiSig(signers, threshold)

activate WI

WI->>SC: initializeMultiSig(walletId, signers, threshold)

activate SC

SC->>CM: deployMultiSigContract(config)

activate CM

CM->>BC: deployContract(bytecode, params)

activate BC

BC-->>CM: contractAddress

deactivate BC

CM-->>SC: deploymentResult

deactivate CM

SC->>SC: updateSecuritySettings()

SC->>NM: notifySigners(signers, config)

activate NM

NM-->>SC: notificationStatus

deactivate NM

SC-->>WI: setupResult

deactivate SC

WI-->>Owner: displayConfirmation()

deactivate WI

sequenceDiagram

actor User

participant WI as WalletInterface  
participant DC as DAppController  
participant PM as PermissionManager  
participant CM as ConnectionManager  
participant DApp

User->>WI: connectToDApp(dappUrl)  
activate WI  
WI->>DC: initializeConnection(dappUrl)  
activate DC

DC->>DApp: requestConnection()  
activate DApp  
DApp-->>DC: connectionRequest  
deactivate DApp

DC->>PM: validatePermissions(permissions)  
activate PM  
PM-->>DC: validationResult  
deactivate PM

DC-->>WI: showPermissionRequest  
WI-->>User: displayPermissions()

User->>WI: approveConnection()

WI->>DC: establishConnection()

DC->>CM: createConnection(dappUrl, permissions)

activate CM

CM-->>DC: connectionDetails

deactivate CM

DC->>DApp: confirmConnection(connectionDetails)

activate DApp

DApp-->>DC: connectionConfirmed

deactivate DApp

DC-->>WI: connectionStatus

deactivate DC

WI-->>User: displayResult()

deactivate WI

sequenceDiagram

participant TM as TransactionMonitor

participant BC as BlockchainConnector

participant NM as NotificationManager

participant DS as DataStore

participant WI as WalletInterface

loop Every Block

TM->>BC: checkTransactionStatus(txHash)

activate BC

BC-->>TM: statusUpdate

deactivate BC

alt Transaction Confirmed

TM->>DS: updateTransactionRecord(txHash, status)

activate DS

DS-->>TM: updateConfirmed

deactivate DS

TM->>NM: sendNotification(CONFIRMED)

activate NM

NM->>WI: updateUI(txStatus)

deactivate NM

else Transaction Failed

TM->>DS: updateTransactionRecord(txHash, FAILED)

activate DS

DS-->>TM: updateConfirmed

deactivate DS

TM->>NM: sendNotification(FAILED)

activate NM

NM->>WI: updateUI(txStatus)

deactivate NM

else Still Pending

```
 TM->>DS: updateConfirmations(txHash)

 activate DS

 DS-->>TM: updateConfirmed

 deactivate DS

end

end
```

## # Sequence Diagram Details

### ## 1. Transfer Assets Sequence

#### ### Components

- **WalletInterface**: User interface layer
- **TransactionController**: Manages transaction lifecycle
- **ValidationManager**: Validates transaction parameters
- **GasEstimator**: Calculates transaction fees
- **BlockchainConnector**: Interfaces with blockchain

#### ### Key Interactions

1. Transaction initiation and validation
2. Balance checking
3. Gas fee estimation
4. User confirmation
5. Transaction broadcast

#### ### Error Handling

- Validation failures



- Insufficient balance
- Network issues
- Gas estimation errors

## ## 2. Setup Multi-Signature Sequence

### ### Components

- **SecurityController**: Manages security features
- **ContractManager**: Handles smart contract deployment
- **NotificationManager**: Manages communications
- **BlockchainConnector**: Blockchain interface

### ### Key Interactions

1. Multi-sig configuration
2. Contract deployment
3. Security settings update
4. Signer notification

### ### Error Handling

- Deployment failures
- Invalid configurations
- Network issues
- Notification failures

## ## 3. DApp Connection Sequence

### ### Components

- **DAppController**: Manages DApp interactions

- **PermissionManager**: Handles permission validation
- **ConnectionManager**: Manages active connections
- **DApp**: External decentralized application

### ### Key Interactions

1. Connection request
2. Permission validation
3. User approval
4. Connection establishment

### ### Error Handling

- Connection failures
- Permission denials
- Timeout issues
- Invalid DApp responses

## ## 4. Transaction Monitoring Sequence

### ### Components

- **TransactionMonitor**: Monitors transaction status
- **BlockchainConnector**: Blockchain interface
- **NotificationManager**: Handles notifications
- **DataStore**: Persistent storage
- **WalletInterface**: User interface updates

### ### Key Interactions

1. Status checking

2. Record updating
3. Notification sending
4. UI updates

### ### Error Handling

- Network failures
- Invalid status updates
- Database errors
- Notification failures

## ## Implementation Considerations

### ### 1. Asynchronous Operations

- Non-blocking blockchain calls
- Background monitoring
- UI responsiveness

### ### 2. Security

- Secure communication channels
- Data encryption
- Permission validation
- Error masking

### ### 3. Performance

- Efficient status polling
- Batch updates

- Connection pooling
- Cache management

#### ### 4. Reliability

- Retry mechanisms
- Fallback options
- Transaction recovery
- State consistency

#### ### 5. Scalability

- Connection management
- Resource pooling
- Load distribution
- Queue management

The sequence diagrams above illustrate the key interactions for four critical system operations:

1. **Transfer Assets:** Shows the complete flow of asset transfer, from initiation to confirmation
2. **Setup Multi-Signature:** Details the process of configuring multi-signature security
3. **DApp Connection:** Illustrates the connection flow between wallet and decentralized applications
4. **Transaction Monitoring:** Shows the continuous monitoring of transaction status

Each diagram includes:

- Clear component responsibilities
- Synchronous and asynchronous operations
- Error handling paths

- State transitions

Would you like me to:

1. Add more detail to any particular sequence?
2. Create diagrams for additional operations?
3. Elaborate on specific component interactions?
4. Provide more implementation details?

## **User**

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

## **AI Model**

I'll create detailed design class diagrams showing the classes, interfaces, and relationships based on the domain model and sequence diagrams.

classDiagram

%% Controllers

```
class WalletController {
 -walletManager: WalletManager
 -securityController: SecurityController
 -transactionController: TransactionController
 +createWallet(config: WalletConfig): Wallet
 +getWalletBalance(address: String): Balance
 +connectToDApp(url: String): Connection
}
```

```
class TransactionController {
 -blockchainConnector: BlockchainConnector
```

```

 -gasEstimator: GasEstimator
 -validationManager: ValidationManager
 +createTransaction(details: TransactionDetails): Transaction
 +signTransaction(tx: Transaction): SignedTransaction
 +broadcastTransaction(tx: SignedTransaction): String
 +estimateGas(tx: TransactionDetails): GasFee
}

```

```

class SecurityController {
 -contractManager: ContractManager
 -notificationManager: NotificationManager
 +setupMultiSig(config: MultiSigConfig): void
 +configureRecovery(config: RecoveryConfig): void
 +validatePermissions(perms: List~Permission~): boolean
}

```

%% Managers

```

class WalletManager {
 -walletRepository: WalletRepository
 -securityManager: SecurityManager
 +createWallet(owner: Address): Wallet
 +getWallet(address: String): Wallet
 +updateWallet(wallet: Wallet): void
 +deleteWallet(address: String): void
}

```

```
class SecurityManager {
 -keyStore: KeyStore
 -encryptionService: EncryptionService
 +generateKeys(): KeyPair
 +encryptData(data: String): String
 +decryptData(data: String): String
 +validateSignature(sig: Signature): boolean
}
```

```
class ValidationManager {
 +validateAddress(address: String): boolean
 +validateTransaction(tx: Transaction): boolean
 +validateBalance(address: String, amount: BigDecimal): boolean
}
```

%% Core Domain Classes

```
class Wallet {
 -address: String
 -owner: Address
 -balance: Balance
 -securitySettings: SecuritySettings
 -status: WalletStatus
 +getBalance(): Balance
 +updateBalance(amount: BigDecimal): void
 +getTransactions(): List~Transaction~
}
```

```
class Transaction {
 -id: String
 -from: Address
 -to: Address
 -amount: BigDecimal
 -status: TransactionStatus
 -gasFee: GasFee
 -timestamp: DateTime
 +sign(key: PrivateKey): void
 +verify(): boolean
 +getStatus(): TransactionStatus
}
```

```
class SecuritySettings {
 -multiSigConfig: MultiSigConfig
 -recoveryConfig: RecoveryConfig
 -spendingLimits: List~SpendingLimit~
 +isMultiSigRequired(): boolean
 +validateSpendingLimit(amount: BigDecimal): boolean
 +getRecoveryOptions(): List~RecoveryOption~
}
```

%% Services

```
class BlockchainConnector {
 -networkConfig: NetworkConfig
```



```
-connectionPool: ConnectionPool

+sendTransaction(tx: SignedTransaction): String

+getTransactionStatus(hash: String): TransactionStatus

+getBalance(address: String): BigDecimal
}
```

```
class ContractManager {

 -blockchainConnector: BlockchainConnector

 +deployContract(bytecode: String, params: Object[]): String

 +callContract(address: String, method: String, params: Object[]): Object

 +getContractEvents(address: String): List~Event~
}
```

```
class GasEstimator {

 -blockchainConnector: BlockchainConnector

 +estimateGas(tx: TransactionDetails): GasFee

 +getGasPrice(): BigDecimal

 +optimizeGasFee(): GasFee
}
```

%% Data Classes

```
class MultiSigConfig {

 -threshold: int

 -signers: List~Address~

 -timelock: Duration

 +addSigner(address: Address): void
```

```
+removeSigner(address: Address): void
+isSignatureRequired(address: Address): boolean
}
```

```
class RecoveryConfig {
 -method: RecoveryMethod
 -guardians: List~Address~
 -cooldown: Duration
 +addGuardian(address: Address): void
 +removeGuardian(address: Address): void
 +initiateRecovery(): void
}
```

```
class GasFee {
 -baseGas: BigDecimal
 -maxGas: BigDecimal
 -priorityFee: BigDecimal
 +calculateTotal(): BigDecimal
 +isWithinLimit(limit: BigDecimal): boolean
}
```

%% Interfaces

```
class IWalletRepository {
 <<interface>>
 +save(wallet: Wallet): void
 +find(address: String): Wallet
```

```
+delete(address: String): void
+update(wallet: Wallet): void
}
```

```
class IBlockchainService {
 <<interface>>
 +broadcast(tx: SignedTransaction): String
 +getStatus(hash: String): TransactionStatus
 +getBalance(address: String): BigDecimal
}
```

```
class ISecurityService {
 <<interface>>
 +encrypt(data: String): String
 +decrypt(data: String): String
 +sign(data: String): Signature
 +verify(data: String, sig: Signature): boolean
}
```

%% Relationships

WalletController --> WalletManager

WalletController --> SecurityController

WalletController --> TransactionController

TransactionController --> BlockchainConnector

TransactionController --> GasEstimator

TransactionController --> ValidationManager

SecurityController --> ContractManager

SecurityController --> SecurityManager

WalletManager --> IWalletRepository

WalletManager --> SecurityManager

Wallet --> SecuritySettings

Wallet --> Transaction

SecuritySettings --> MultiSigConfig

SecuritySettings --> RecoveryConfig

Transaction --> GasFee

BlockchainConnector ..|> IBlockchainService

SecurityManager ..|> ISecurityService

## # Smart Wallet System Design Class Details

### ## 1. Controller Layer

#### ### WalletController

Primary interface for wallet operations

- Manages wallet lifecycle

- Coordinates between components
- Handles high-level operations

### ### TransactionController

Manages transaction operations

- Transaction creation and validation
- Gas estimation
- Broadcasting to blockchain

### ### SecurityController

Handles security-related operations

- Multi-signature setup
- Recovery configuration
- Permission management

## ## 2. Manager Layer

### ### WalletManager

Core wallet business logic

- Wallet creation and management
- Balance tracking
- State management

### ### SecurityManager

Security operations handler

- Key management

- Encryption/Decryption
- Signature validation

### ### ValidationManager

Input and state validation

- Address validation
- Transaction validation
- Balance verification

## ## 3. Core Domain Classes

### ### Wallet

Central domain entity

- Holds wallet state
- Manages assets
- Tracks transactions

### ### Transaction

Represents blockchain transactions

- Transaction details
- Status tracking
- Gas fee management

### ### SecuritySettings

Security configuration container

- Multi-sig settings

- Recovery options
- Spending limits

## ## 4. Services

### ### BlockchainConnector

Blockchain interaction service

- Network communication
- Transaction broadcasting
- State queries

### ### ContractManager

Smart contract operations

- Contract deployment
- Method execution
- Event handling

### ### GasEstimator

Gas fee management

- Fee estimation
- Price optimization
- Limit calculations

## ## 5. Data Classes

### ### MultiSigConfig

## Multi-signature settings

- Signer management
- Threshold configuration
- Timelock settings

## ### RecoveryConfig

### Recovery mechanism settings

- Guardian management
- Recovery methods
- Cooldown periods

## ### GasFee

### Gas fee calculations

- Base fee
- Priority fee
- Total calculations

## ## 6. Interfaces

## ### IWalletRepository

### Data persistence interface

- CRUD operations
- Query operations
- State persistence

## ### IBlockchainService



Blockchain service interface

- Network operations
- Status queries
- Balance checks

### ISecurityService

Security operations interface

- Encryption operations
- Signing operations
- Verification methods

## Design Principles Applied

1. **SOLID Principles**

- Single Responsibility Principle
- Open/Closed Principle
- Interface Segregation
- Dependency Inversion

2. **Design Patterns**

- Factory Pattern for creation
- Strategy Pattern for operations
- Observer Pattern for events
- Repository Pattern for data

3. **Security Considerations**

- Encapsulation of sensitive data
- Separation of concerns
- Principle of least privilege

#### 4. **Extensibility**

- Interface-based design
- Plugin architecture
- Configurable components

### ## Implementation Guidelines

#### 1. **Error Handling**

- Custom exceptions
- Error propagation
- Logging strategy

#### 2. **Transaction Management**

- ACID properties
- State consistency
- Rollback mechanisms

#### 3. **Performance Optimization**

- Caching strategy
- Connection pooling
- Batch processing

#### 4. **\*\*Security Implementation\*\***

- Encryption standards
- Key management
- Access control

The design class diagram and details above show:

##### 1. **Layered Architecture:**

- Controllers for operation coordination
- Managers for business logic
- Services for external interactions
- Domain classes for core entities

##### 2. **Design Patterns:**

- Factory pattern for creation
- Strategy pattern for operations
- Observer pattern for events
- Repository pattern for data access

##### 3. **SOLID Principles:**

- Single responsibility
- Interface segregation
- Dependency inversion

Would you like me to:

1. Add more detail to specific classes?
2. Explain particular design decisions?
3. Show additional relationships or patterns?
4. Provide implementation guidelines for specific components?

## User

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

## AI Model

I'll create a Java implementation of the smart wallet system, focusing on the core components defined in the design.

```
// Core domain entities
```

```
package com.smartwallet.domain;
```

```
import java.math.BigDecimal;
```

```
import java.time.LocalDateTime;
```

```
import java.util.ArrayList;
```

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

```
import java.util.UUID;
```

```
public class Wallet {
```

```
 private final String address;
```

```
 private final Address owner;
```

```
 private Balance balance;
```

```
 private SecuritySettings securitySettings;
```

```
 private WalletStatus status;
```

```
 private List<Transaction> transactions;
```

```
 public Wallet(String address, Address owner) {
```

```
 this.address = address;
 this.owner = owner;
 this.balance = new Balance();
 this.securitySettings = new SecuritySettings();
 this.status = WalletStatus.ACTIVE;
 this.transactions = new ArrayList<>();
}
```

```
public Balance getBalance() {
 return balance;
}
```

```
public void updateBalance(BigDecimal amount) {
 this.balance.update(amount);
}
```

```
public List<Transaction> getTransactions() {
 return new ArrayList<>(transactions);
}
```

```
public void addTransaction(Transaction transaction) {
 this.transactions.add(transaction);
}
```

```
public SecuritySettings getSecuritySettings() {
 return securitySettings;
}
```

```

 }

 public WalletStatus getStatus() {
 return status;
 }

 public void setStatus(WalletStatus status) {
 this.status = status;
 }
}

public class Transaction {
 private final String id;
 private final Address from;
 private final Address to;
 private final BigDecimal amount;
 private TransactionStatus status;
 private final GasFee gasFee;
 private final LocalDateTime timestamp;
 private byte[] signature;

 public Transaction(Address from, Address to, BigDecimal amount, GasFee gasFee) {
 this.id = UUID.randomUUID().toString();
 this.from = from;
 this.to = to;
 this.amount = amount;
 }

```

```
 this.gasFee = gasFee;

 this.status = TransactionStatus.PENDING;

 this.timestamp = LocalDateTime.now();
}
```

```
public void sign(PrivateKey privateKey) {

 // Implementation of transaction signing

 byte[] message = getSignatureMessage();

 this.signature = SignatureUtils.sign(message, privateKey);
}
```

```
public boolean verify() {

 if (signature == null) {

 return false;

 }

 byte[] message = getSignatureMessage();

 return SignatureUtils.verify(message, signature, from.getPublicKey());
}
```

```
private byte[] getSignatureMessage() {

 // Combine transaction data for signing

 return SignatureUtils.combineForSignature(

 from.toString(),

 to.toString(),

 amount.toString(),

 timestamp.toString()
);
}
```

```

);
}

// Getters

public String getId() { return id; }

public Address getFrom() { return from; }

public Address getTo() { return to; }

public BigDecimal getAmount() { return amount; }

public TransactionStatus getStatus() { return status; }

public GasFee getGasFee() { return gasFee; }

public LocalDateTime getTimestamp() { return timestamp; }
}

public class SecuritySettings {

 private MultiSigConfig multiSigConfig;

 private RecoveryConfig recoveryConfig;

 private List<SpendingLimit> spendingLimits;

 public SecuritySettings() {

 this.spendingLimits = new ArrayList<>();

 }

 public boolean isMultiSigRequired() {

 return multiSigConfig != null && multiSigConfig.isEnabled();

 }
}

```



```

public boolean validateSpendingLimit(BigDecimal amount) {
 return spendingLimits.stream()
 .allMatch(limit -> limit.isWithinLimit(amount));
}

```

```

public List<RecoveryOption> getRecoveryOptions() {
 return recoveryConfig != null
 ? recoveryConfig.getAvailableOptions()
 : new ArrayList<>();
}

```

// Getters and setters

```

public MultiSigConfig getMultiSigConfig() { return multiSigConfig; }
public void setMultiSigConfig(MultiSigConfig config) { this.multiSigConfig = config; }
public RecoveryConfig getRecoveryConfig() { return recoveryConfig; }
public void setRecoveryConfig(RecoveryConfig config) { this.recoveryConfig = config; }
}

```

```

public class Balance {
 private Map<String, BigDecimal> assetBalances;

 public Balance() {
 this.assetBalances = new HashMap<>();
 }
}

```

```

public void update(String asset, BigDecimal amount) {

```

```

 assetBalances.compute(asset, (k, v) ->
 (v == null) ? amount : v.add(amount));
 }

 public BigDecimal getBalance(String asset) {
 return assetBalances.getDefault(asset, BigDecimal.ZERO);
 }

 public Map<String, BigDecimal> getAllBalances() {
 return new HashMap<>(assetBalances);
 }
}

public enum WalletStatus {
 ACTIVE,
 LOCKED,
 RECOVERY_PENDING,
 DISABLED
}

public enum TransactionStatus {
 PENDING,
 CONFIRMED,
 FAILED,
 REJECTED
}

```

```
package com.smartwallet.controller;
```

```
import com.smartwallet.domain.*;
```

```
import com.smartwallet.service.*;
```

```
import com.smartwallet.exception.*;
```

```
public class WalletController {
```

```
 private final WalletManager walletManager;
```

```
 private final SecurityController securityController;
```

```
 private final TransactionController transactionController;
```

```
 public WalletController(
```

```
 WalletManager walletManager,
```

```
 SecurityController securityController,
```

```
 TransactionController transactionController
```

```
){
```

```
 this.walletManager = walletManager;
```

```
 this.securityController = securityController;
```

```
 this.transactionController = transactionController;
```

```
 }
```

```
 public Wallet createWallet(WalletConfig config) throws WalletException {
```

```
 validateWalletConfig(config);
```

```
 return walletManager.createWallet(config.getOwner());
```

```
 }
```

```

public Balance getWalletBalance(String address) throws WalletException {
 Wallet wallet = walletManager.getWallet(address);
 if (wallet == null) {
 throw new WalletNotFoundException("Wallet not found: " + address);
 }
 return wallet.getBalance();
}

```

```

public Connection connectToDApp(String walletAddress, String dappUrl)
 throws WalletException {
 Wallet wallet = walletManager.getWallet(walletAddress);
 if (wallet == null) {
 throw new WalletNotFoundException("Wallet not found: " + walletAddress);
 }
 // Implement DApp connection logic
 return new Connection(wallet, dappUrl);
}

```

```

private void validateWalletConfig(WalletConfig config) throws WalletException {
 if (config == null || config.getOwner() == null) {
 throw new InvalidConfigurationException("Invalid wallet configuration");
 }
}
}

```

```

public class TransactionController {

```

```
private final BlockchainConnector blockchainConnector;
private final GasEstimator gasEstimator;
private final ValidationManager validationManager;
```

```
public TransactionController(
 BlockchainConnector blockchainConnector,
 GasEstimator gasEstimator,
 ValidationManager validationManager
) {
 this.blockchainConnector = blockchainConnector;
 this.gasEstimator = gasEstimator;
 this.validationManager = validationManager;
}
```

```
public Transaction createTransaction(TransactionDetails details)
 throws TransactionException {
 validateTransactionDetails(details);

 GasFee gasFee = gasEstimator.estimateGas(details);
 Transaction transaction = new Transaction(
 details.getFrom(),
 details.getTo(),
 details.getAmount(),
 gasFee
);
```

```
 return transaction;
}
```

```
public SignedTransaction signTransaction(Transaction transaction, PrivateKey privateKey)
 throws TransactionException {
 if (!validationManager.validateTransaction(transaction)) {
 throw new InvalidTransactionException("Invalid transaction");
 }
```

```
 transaction.sign(privateKey);
 return new SignedTransaction(transaction);
}
```

```
public String broadcastTransaction(SignedTransaction signedTransaction)
 throws TransactionException {
 try {
 return blockchainConnector.sendTransaction(signedTransaction);
 } catch (BlockchainException e) {
 throw new TransactionException("Failed to broadcast transaction", e);
 }
}
```

```
private void validateTransactionDetails(TransactionDetails details)
 throws TransactionException {
 if (!validationManager.validateAddress(details.getTo().toString())) {
 throw new InvalidAddressException("Invalid recipient address");
 }
```

```

 }

 if (!validationManager.validateBalance(
 details.getFrom().toString(),
 details.getAmount())) {
 throw new InsufficientBalanceException("Insufficient balance");
 }
}
}
}

```

```

public class SecurityController {

 private final ContractManager contractManager;
 private final NotificationManager notificationManager;
 private final SecurityManager securityManager;

 public SecurityController(
 ContractManager contractManager,
 NotificationManager notificationManager,
 SecurityManager securityManager
){
 this.contractManager = contractManager;
 this.notificationManager = notificationManager;
 this.securityManager = securityManager;
 }

 public void setupMultiSig(String walletAddress, MultiSigConfig config)
 throws SecurityException {

```

```

validateMultiSigConfig(config);

try {
 // Deploy multi-sig contract

 String contractAddress = contractManager.deployContract(
 "MultiSig",
 config.getDeploymentParams()
);

 // Update wallet security settings
 config.setContractAddress(contractAddress);
 updateWalletSecuritySettings(walletAddress, config);

 // Notify signers
 notifySigners(config.getSigners());

} catch (ContractException e) {
 throw new SecurityException("Failed to setup multi-sig", e);
}

}

public void configureRecovery(String walletAddress, RecoveryConfig config)
 throws SecurityException {
 validateRecoveryConfig(config);

 try {

```



```

// Deploy recovery contract if needed
if (config.requiresContract()) {
 String contractAddress = contractManager.deployContract(
 "Recovery",
 config.getDeploymentParams()
);
 config.setContractAddress(contractAddress);
}

// Update wallet security settings
updateWalletSecuritySettings(walletAddress, config);

// Notify guardians
notifyGuardians(config.getGuardians());

} catch (ContractException e) {
 throw new SecurityException("Failed to configure recovery", e);
}
}

private void validateMultiSigConfig(MultiSigConfig config) throws SecurityException {
 if (config == null ||
 config.getSigners().isEmpty() ||
 config.getThreshold() <= 0 ||
 config.getThreshold() > config.getSigners().size()) {
 throw new InvalidConfigurationException("Invalid multi-sig configuration");
 }
}

```

```
}
}
```

```
private void validateRecoveryConfig(RecoveryConfig config) throws SecurityException {
 if (config == null ||
 config.getGuardians().isEmpty() ||
 config.getMethod() == null) {
 throw new InvalidConfigurationException("Invalid recovery configuration");
 }
}
```

```
private void updateWalletSecuritySettings(String walletAddress, Object config)
 throws SecurityException {
 // Implementation of security settings update
}
```

```
private void notifySigners(List<Address> signers) {
 signers.forEach(signer ->
 notificationManager.sendNotification(
 signer,
 "Multi-sig Configuration",
 "You have been added as a signer"
)
);
}
```

```

private void notifyGuardians(List<Address> guardians) {
 guardians.forEach(guardian ->
 notificationManager.sendNotification(
 guardian,
 "Recovery Configuration",
 "You have been added as a guardian"
)
);
}
}

package com.smartwallet.service;

import com.smartwallet.domain.*;
import com.smartwallet.exception.*;

public class BlockchainConnector implements IBlockchainService {
 private final NetworkConfig networkConfig;
 private final ConnectionPool connectionPool;

 public BlockchainConnector(NetworkConfig networkConfig) {
 this.networkConfig = networkConfig;
 this.connectionPool = new ConnectionPool(networkConfig);
 }

 @Override
 public String sendTransaction(SignedTransaction tx) throws BlockchainException {

```

```

try {
 Web3Connection connection = connectionPool.acquire();
 try {
 String hash = connection.sendRawTransaction(tx.getEncoded());
 return hash;
 } finally {
 connectionPool.release(connection);
 }
} catch (Exception e) {
 throw new BlockchainException("Failed to send transaction", e);
}
}

```

@Override

```

public TransactionStatus getTransactionStatus(String hash)
 throws BlockchainException {
 try {
 Web3Connection connection = connectionPool.acquire();
 try {
 TransactionReceipt receipt = connection.getTransactionReceipt(hash);
 return mapToTransactionStatus(receipt);
 } finally {
 connectionPool.release(connection);
 }
 } catch (Exception e) {
 throw new BlockchainException("Failed to get transaction status", e);
 }
}

```

```
}
}
```

@Override

```
public BigDecimal getBalance(String address) throws BlockchainException {
```

```
 try {
```

```
 Web3Connection connection = connectionPool.acquire();
```

```
 try {
```

```
 return connection.getBalance(address);
```

```
 } finally {
```

```
 connectionPool.release(connection);
```

```
 }
```

```
 } catch (Exception e) {
```

```
 throw new BlockchainException("Failed to get balance", e);
```

```
 }
```

```
}
```

```
private TransactionStatus mapToTransactionStatus(TransactionReceipt receipt) {
```

```
 if (receipt == null) {
```

```
 return TransactionStatus.PENDING;
```

```
 }
```

```
 return receipt.isSuccessful() ?
```

```
 TransactionStatus.CONFIRMED : TransactionStatus.FAILED;
```

```
}
```

```
}
```

```

public class ContractManager {

 private final BlockchainConnector blockchainConnector;

 private final ContractLoader contractLoader;

 public ContractManager(

 BlockchainConnector blockchainConnector,

 ContractLoader contractLoader

){

 this.blockchainConnector = blockchainConnector;

 this.contractLoader = contractLoader;

 }

 public String deployContract(String contractName, Object[] params)

 throws ContractException {

 try {

 // Load contract bytecode and ABI

 ContractDefinition contractDef = contractLoader.loadContract(contractName);

 // Encode constructor parameters

 byte[] constructorData = AbiEncoder.encodeConstructor(

 contractDef.getAbi(),

 params

);

 // Combine bytecode and constructor data

 byte[] deployData = ByteUtils.concat(

```

```

 contractDef.getBytecode(),
 constructorData
);

 // Create and send deployment transaction
 Transaction tx = new Transaction(
 null, // Contract creation has no 'to' address
 BigDecimal.ZERO,

```

## **User**

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

## **AI Model**

I'll create a comprehensive test suite for the smart wallet system, including unit tests, integration tests, and system tests.

```

// Unit Tests for core components
package com.smartwallet.test.unit;

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

public class WalletControllerTest {
 @Mock
 private WalletManager walletManager;
 @Mock

```

```
private SecurityController securityController;
```

```
@Mock
```

```
private TransactionController transactionController;
```

```
private WalletController walletController;
```

```
@BeforeEach
```

```
void setUp() {
```

```
 MockitoAnnotations.openMocks(this);
```

```
 walletController = new WalletController(
```

```
 walletManager,
```

```
 securityController,
```

```
 transactionController
```

```
);
```

```
}
```

```
@Test
```

```
void createWallet_ValidConfig_Success() {
```

```
 // Arrange
```

```
 WalletConfig config = new WalletConfig(new Address("0x123"));
```

```
 Wallet expectedWallet = new Wallet("0x456", config.getOwner());
```

```
 when(walletManager.createWallet(config.getOwner()))
```

```
 .thenReturn(expectedWallet);
```

```
 // Act
```

```
 Wallet result = walletController.createWallet(config);
```



```
// Assert

assertNotNull(result);

assertEquals(expectedWallet.getAddress(), result.getAddress());

verify(walletManager).createWallet(config.getOwner());

}
```

@Test

```
void createWallet_NullConfig_ThrowsException() {

 // Act & Assert

 assertThrows(InvalidConfigurationException.class,

 () -> walletController.createWallet(null));

}
```

@Test

```
void getWalletBalance_ExistingWallet_ReturnsBalance() {

 // Arrange

 String address = "0x123";

 Wallet mockWallet = mock(Wallet.class);

 Balance expectedBalance = new Balance();

 when(walletManager.getWallet(address)).thenReturn(mockWallet);

 when(mockWallet.getBalance()).thenReturn(expectedBalance);

 // Act

 Balance result = walletController.getWalletBalance(address);
```

```
 // Assert
 assertNotNull(result);
 assertEquals(expectedBalance, result);
 verify(walletManager).getWallet(address);
 }
}
```

```
public class TransactionControllerTest {

 @Mock
 private BlockchainConnector blockchainConnector;

 @Mock
 private GasEstimator gasEstimator;

 @Mock
 private ValidationManager validationManager;

 private TransactionController transactionController;

 @BeforeEach
 void setUp() {
 MockitoAnnotations.openMocks(this);
 transactionController = new TransactionController(
 blockchainConnector,
 gasEstimator,
 validationManager
);
 }
}
```

@Test

void createTransaction\_ValidDetails\_Success() {

    // Arrange

    TransactionDetails details = createValidTransactionDetails();

    GasFee expectedGasFee = new GasFee(BigDecimal.ONE);

    when(gasEstimator.estimateGas(details)).thenReturn(expectedGasFee);

    when(validationManager.validateAddress(anyString())).thenReturn(true);

    when(validationManager.validateBalance(anyString(), any())).thenReturn(true);

    // Act

    Transaction result = transactionController.createTransaction(details);

    // Assert

    assertNotNull(result);

    assertEquals(details.getFrom(), result.getFrom());

    assertEquals(details.getTo(), result.getTo());

    assertEquals(details.getAmount(), result.getAmount());

    assertEquals(expectedGasFee, result.getGasFee());

}

@Test

void signTransaction\_ValidTransaction\_Success() {

    // Arrange

    Transaction transaction = createValidTransaction();

    PrivateKey privateKey = new PrivateKey("0xkey");

```
when(validationManager.validateTransaction(transaction)).thenReturn(true);

// Act
SignedTransaction result = transactionController.signTransaction(
 transaction,
 privateKey
);

// Assert
assertNotNull(result);
assertTrue(result.getTransaction().verify());
}
}
```

```
public class SecuritySettingsTest {
 private SecuritySettings securitySettings;

 @BeforeEach
 void setUp() {
 securitySettings = new SecuritySettings();
 }
}
```

```
@Test
void isMultiSigRequired_WithEnabledConfig_ReturnsTrue() {
 // Arrange
 MultiSigConfig config = new MultiSigConfig();
}
```

```
 config.setEnabled(true);
 securitySettings.setMultiSigConfig(config);

 // Act
 boolean result = securitySettings.isMultiSigRequired();

 // Assert
 assertTrue(result);
}

@Test
void validateSpendingLimit_WithinLimit_ReturnsTrue() {
 // Arrange
 SpendingLimit limit = new SpendingLimit(
 BigDecimal.TEN,
 Duration.ofDays(1)
);
 securitySettings.addSpendingLimit(limit);

 // Act
 boolean result = securitySettings.validateSpendingLimit(BigDecimal.ONE);

 // Assert
 assertTrue(result);
}
}
```

```
package com.smartwallet.test.integration;

import org.junit.jupiter.api.*;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.TestPropertySource;
import org.springframework.beans.factory.annotation.Autowired;

@SpringBootTest
@TestPropertySource(locations = "classpath:test.properties")
public class WalletTransactionIntegrationTest {

 @Autowired
 private WalletController walletController;

 @Autowired
 private BlockchainConnector blockchainConnector;

 @Autowired
 private TestDataHelper testDataHelper;

 private Wallet testWallet;

 @BeforeEach
 void setUp() {
 testWallet = testDataHelper.createTestWallet();
 }
}
```

@Test

void completeTransactionFlow\_Success() throws Exception {

    // Arrange

    Address recipient = new Address("0xrecipient");

    BigDecimal amount = BigDecimal.ONE;

    TransactionDetails details = new TransactionDetails(

        testWallet.getAddress(),

        recipient,

        amount

    );

    // Act - Create and sign transaction

    Transaction transaction = walletController.createTransaction(details);

    SignedTransaction signedTx = walletController.signTransaction(

        transaction,

        testDataHelper.getTestPrivateKey()

    );

    // Broadcast transaction

    String txHash = walletController.broadcastTransaction(signedTx);

    // Assert - Check transaction status

    await()

        .atMost(30, TimeUnit.SECONDS)

        .pollInterval(2, TimeUnit.SECONDS)

        .until(() -> {

```

 TransactionStatus status = blockchainConnector
 .getTransactionStatus(txHash);
 return status == TransactionStatus.CONFIRMED;
 });

 // Verify balances
 Balance recipientBalance = walletController
 .getWalletBalance(recipient.toString());
 assertEquals(amount, recipientBalance.getBalance("ETH"));
}
}

```

```

@SpringBootTest
public class SecuritySettingsIntegrationTest {

 @Autowired
 private SecurityController securityController;

 @Autowired
 private ContractManager contractManager;

 @Autowired
 private TestDataHelper testDataHelper;

 private Wallet testWallet;

 @BeforeEach

```



```
void setUp() {
 testWallet = testDataHelper.createTestWallet();
}
```

@Test

```
void setupMultiSig_CompleteFlow_Success() throws Exception {
 // Arrange

 List<Address> signers = testDataHelper.createTestSigners(3);
 MultiSigConfig config = new MultiSigConfig();
 config.setSigners(signers);
 config.setThreshold(2);

 // Act

 securityController.setupMultiSig(testWallet.getAddress(), config);

 // Assert

 String contractAddress = testWallet
 .getSecuritySettings()
 .getMultiSigConfig()
 .getContractAddress();

 // Verify contract deployment

 assertTrue(contractManager.isContractDeployed(contractAddress));

 // Verify contract configuration

 MultiSigContract contract = contractManager
```

```

 .getMultiSigContract(contractAddress);
 assertEquals(2, contract.getThreshold());
 assertTrue(contract.getSigners().containsAll(signers));
}
}

package com.smartwallet.test.system;

```

```

import io.cucumber.java.en.*;
import org.springframework.boot.test.context.SpringBootTest;

```

```

@SpringBootTest
public class WalletOperationsSystemTest {

 @Autowired
 private TestContext testContext;

 @Given("a user with address {string}")
 public void givenUserWithAddress(String address) {
 testContext.setCurrentAddress(new Address(address));
 }
}

```

```

 @When("they create a new wallet")
 public void whenCreateNewWallet() {
 WalletConfig config = new WalletConfig(testContext.getCurrentAddress());
 testContext.setWallet(
 walletController.createWallet(config)
);
 }
}

```

```
}
```

```
@Then("the wallet should be active")
```

```
public void thenWalletShouldBeActive() {
```

```
 Wallet wallet = testContext.getWallet();
```

```
 assertNotNull(wallet);
```

```
 assertEquals(WalletStatus.ACTIVE, wallet.getStatus());
```

```
}
```

```
@When("they transfer {decimal} ETH to {string}")
```

```
public void whenTransferEth(BigDecimal amount, String recipient) {
```

```
 TransactionDetails details = new TransactionDetails(
```

```
 testContext.getCurrentAddress(),
```

```
 new Address(recipient),
```

```
 amount
```

```
);
```

```
 Transaction tx = walletController.createTransaction(details);
```

```
 SignedTransaction signedTx = walletController.signTransaction(
```

```
 tx,
```

```
 testContext.getPrivateKey()
```

```
);
```

```
 String txHash = walletController.broadcastTransaction(signedTx);
```

```
 testContext.setLastTransactionHash(txHash);
```

```
}
```

```
@Then("the transaction should be confirmed")
```

```

public void thenTransactionShouldBeConfirmed() {
 String txHash = testContext.getLastTransactionHash();
 await()
 .atMost(30, TimeUnit.SECONDS)
 .until(() -> {
 TransactionStatus status = blockchainConnector
 .getTransactionStatus(txHash);
 return status == TransactionStatus.CONFIRMED;
 });
}
}

```

```

@CucumberOptions(
 features = "src/test/resources/features",
 glue = "com.smartwallet.test.system"
)
public class SmartWalletSystemTestRunner {
}

```

# src/test/resources/features/wallet\_operations.feature

Feature: Wallet Operations

As a user

I want to manage my digital assets

So that I can participate in blockchain transactions

Scenario: Create new wallet

Given a user with address "0x123456789"

When they create a new wallet  
Then the wallet should be active  
And the wallet should have zero balance

#### Scenario: Transfer ETH between wallets

Given a user with address "0x123456789"  
And they have an active wallet with 2 ETH  
When they transfer 1 ETH to "0x987654321"  
Then the transaction should be confirmed  
And their wallet balance should be 1 ETH  
And the recipient balance should be 1 ETH

#### Scenario: Setup multi-signature wallet

Given a user with address "0x123456789"  
And they have an active wallet  
When they configure multi-sig with:

| signer    | threshold |
|-----------|-----------|
| 0xsigner1 | 2         |
| 0xsigner2 |           |
| 0xsigner3 |           |

Then the multi-sig configuration should be active  
And all signers should be notified

#### Scenario: Execute multi-sig transaction

Given a multi-sig wallet with 2/3 threshold  
And a pending transaction of 1 ETH to "0x987654321"

When 2 signers approve the transaction  
Then the transaction should be confirmed  
And the recipient should receive 1 ETH

```
src/test/resources/application-test.properties

blockchain.network=test

blockchain.node.url=http://localhost:8545

blockchain.gas.limit=6000000

blockchain.gas.price=20000000000
```

This test suite provides comprehensive coverage through:

**1. Unit Tests:**

- Tests individual components in isolation
- Uses mocking for dependencies
- Covers edge cases and error conditions

**2. Integration Tests:**

- Tests component interactions
- Uses actual database and services
- Verifies contract deployments

**3. System Tests:**

- BDD-style tests with Cucumber
- End-to-end scenarios
- Tests complete user workflows

Key test areas include:

**1. Core Functionality:**

- Wallet creation

- Transaction processing
- Balance management

**2. Security Features:**

- Multi-signature setup
- Transaction signing
- Spending limits

**3. Error Handling:**

- Invalid inputs
- Network failures
- Insufficient funds

Would you like me to:

1. Add more test scenarios?
2. Create tests for specific components?
3. Add performance or load tests?
4. Enhance error condition coverage?