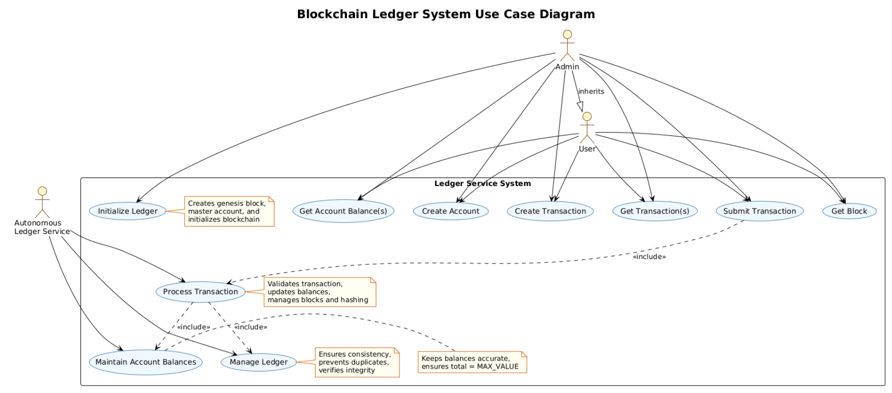
Assignment 1 - 310-1: Smart Store Solution Summary

Author:

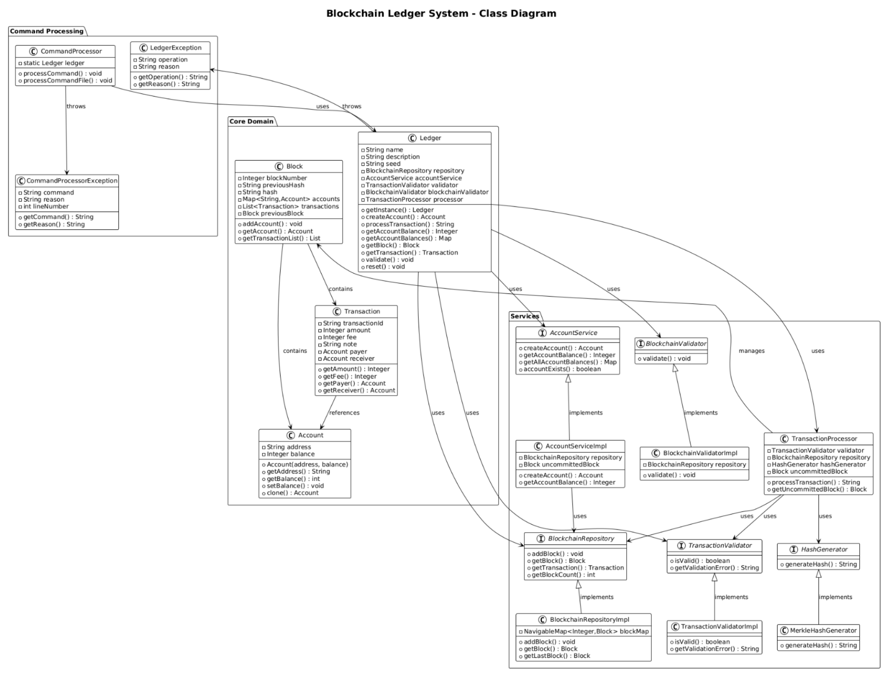
- Name: Tyler Stier
- Group Members: Tyler Stier, Davis Wood
- Email: tstier@chapman.edu, dwood@chapman.edu
- Date: Sep 24, 2025

Technical Design

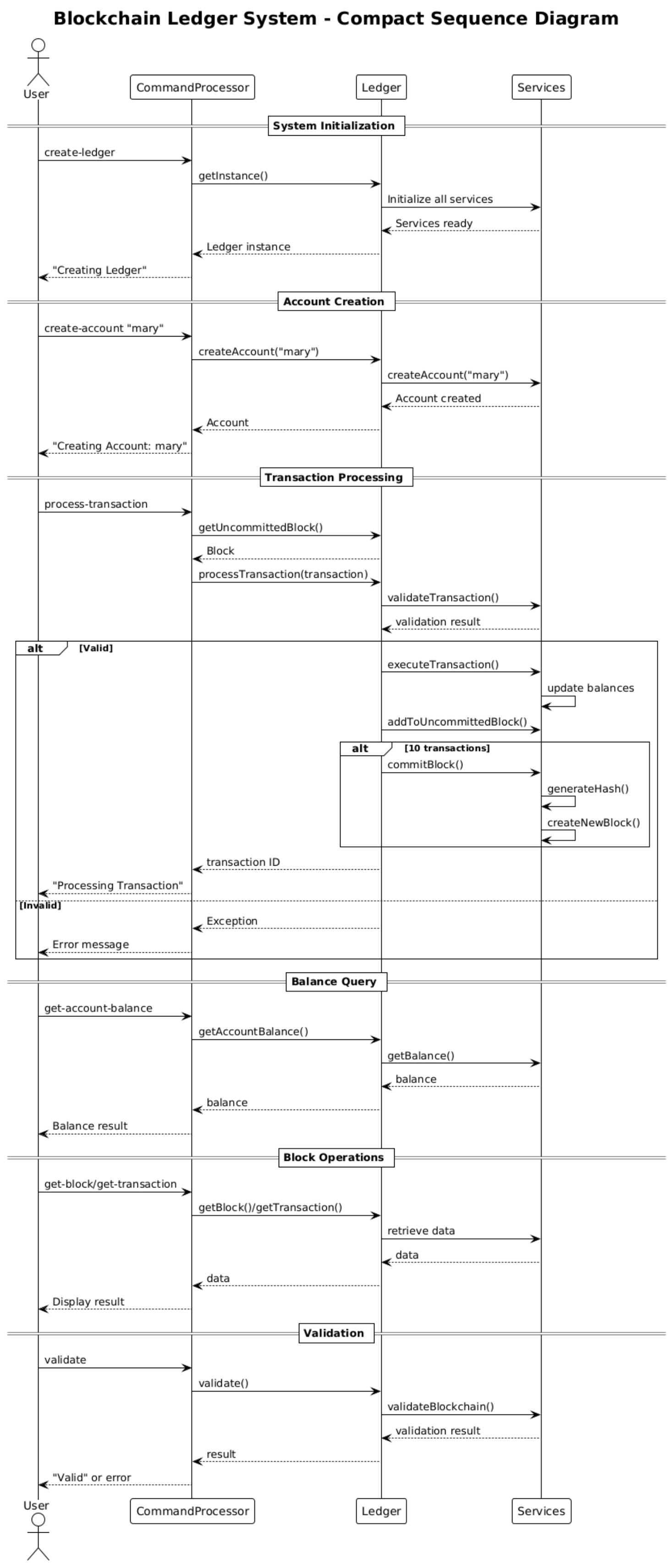
Use Case Diagram


Blockchain Ledger System Use Case Diagram
Admin
inherits
User
Autonomous Ledger Service
Ledger Service System
Initialize Ledger
Creates genesis block, master account, and initializes blockchain
Get Account Balance(s)
Create Account
Create Transaction
Get Transaction(s)
Submit Transaction
Get Block
«include»
Process Transaction
Validates transaction, updates balances, manages blocks and hashing
«include»
«include»
Maintain Account Balances
Manage Ledger
Ensures consistency, prevents duplicates, verifies integrity
Keeps balances accurate, ensures total = MAX_VALUE

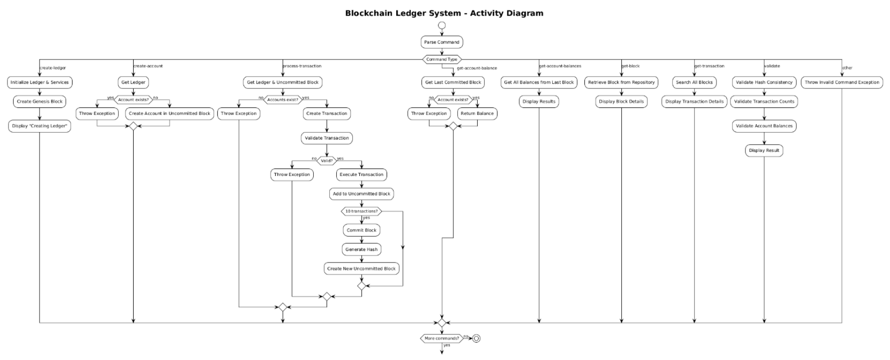
Class Diagram



Sequence Diagram



Activity Diagram

- 

SOLID Design Principles Implementation Description

SRP

The Single Responsibility Principle was implemented by breaking down the monolithic Ledger class into multiple focused services, each with a single responsibility. The original Ledger class was doing too many things: managing accounts, processing transactions, validating blockchain, storing data, and generating hashes. Now it acts as a facade that delegates to appropriate services, with each service having one clear responsibility.
The BlockchainRepository handles only blockchain data storage and retrieval operations, the AccountService manages only account-related operations like creation and balance queries,
the TransactionValidator focuses solely on transaction validation logic,
the BlockchainValidator handles only blockchain integrity validation,
the TransactionProcessor manages only transaction processing and block commitment, and the HashGenerator is responsible only for hash generation using Merkle trees. This separation ensures that each class has a single reason to change and makes the code more maintainable and testable.

OSP

The Open/Closed Principle was implemented through the use of interfaces and dependency injection. The system uses interface-based design where interfaces like BlockchainRepository, AccountService, TransactionValidator, etc. define contracts without implementation details. This makes the system extensible because new implementations can be added without modifying existing code, such as different hash algorithms, validation strategies, or storage mechanisms. The core Ledger class doesn't need to change when new implementations are added, making it closed for modification. For example, you could create
a SHA256HashGenerator or BCryptHashGenerator implementing HashGen erator without touching the Ledger class. The system is now open for extension through new implementations but closed for modification of existing stable code.

LSP

The Liskov Substitution Principle was implemented by ensuring proper inheritance and interface compliance. The Account class was updated to properly implement the Cloneable interface with a correct clone() method that returns Account type instead of Object, ensuring proper behavioral consistency. All concrete implementations
like BlockchainRepositoryImpl, AccountServiceImpl, etc. can be substituted for their interfaces without breaking functionality. Each implementation maintains the same behavioral contracts as defined by their interfaces, and proper exception handling ensures that implementations don't throw unexpected exceptions that would break the substitution principle.
The Account.clone() method now properly implements
the Cloneable interface and can be substituted anywhere an Account is expected.

ISP

The Interface Segregation Principle was implemented by creating small, focused interfaces instead of large, monolithic ones. Each interface is focused on a specific concern, preventing clients from depending on methods they don't use. The BlockchainRepository contains only
data access methods with no validation or processing,
the AccountService handles only account management operations without transaction processing, the TransactionValidator focuses only on validation logic without data access or processing, the BlockchainValidator handles only blockchain integrity checks without account management, and
the HashGenerator is responsible only for hash generation without validation or data access. This segregation means that a class that only needs to validate transactions doesn't need to know about account management or blockchain storage, reducing unnecessary dependencies and making the system more modular.

DIP

The Dependency Inversion Principle was implemented through dependency injection and abstraction dependencies. The Ledger class receives all its dependencies through constructor parameters, depending on abstractions (interfaces) rather than concrete implementations. All services depend on interfaces like BlockchainRepository, AccountService, etc. rather than concrete classes, creating loose coupling throughout the system. The Ledger.getInstance() method creates and injects dependencies, inverting the control of dependency creation. The high-level Ledger class now depends on abstractions (interfaces) rather than low-level concrete implementations, making the system more flexible and testable. This approach allows for easy testing with mock implementations and makes the system more maintainable by reducing tight coupling between components.