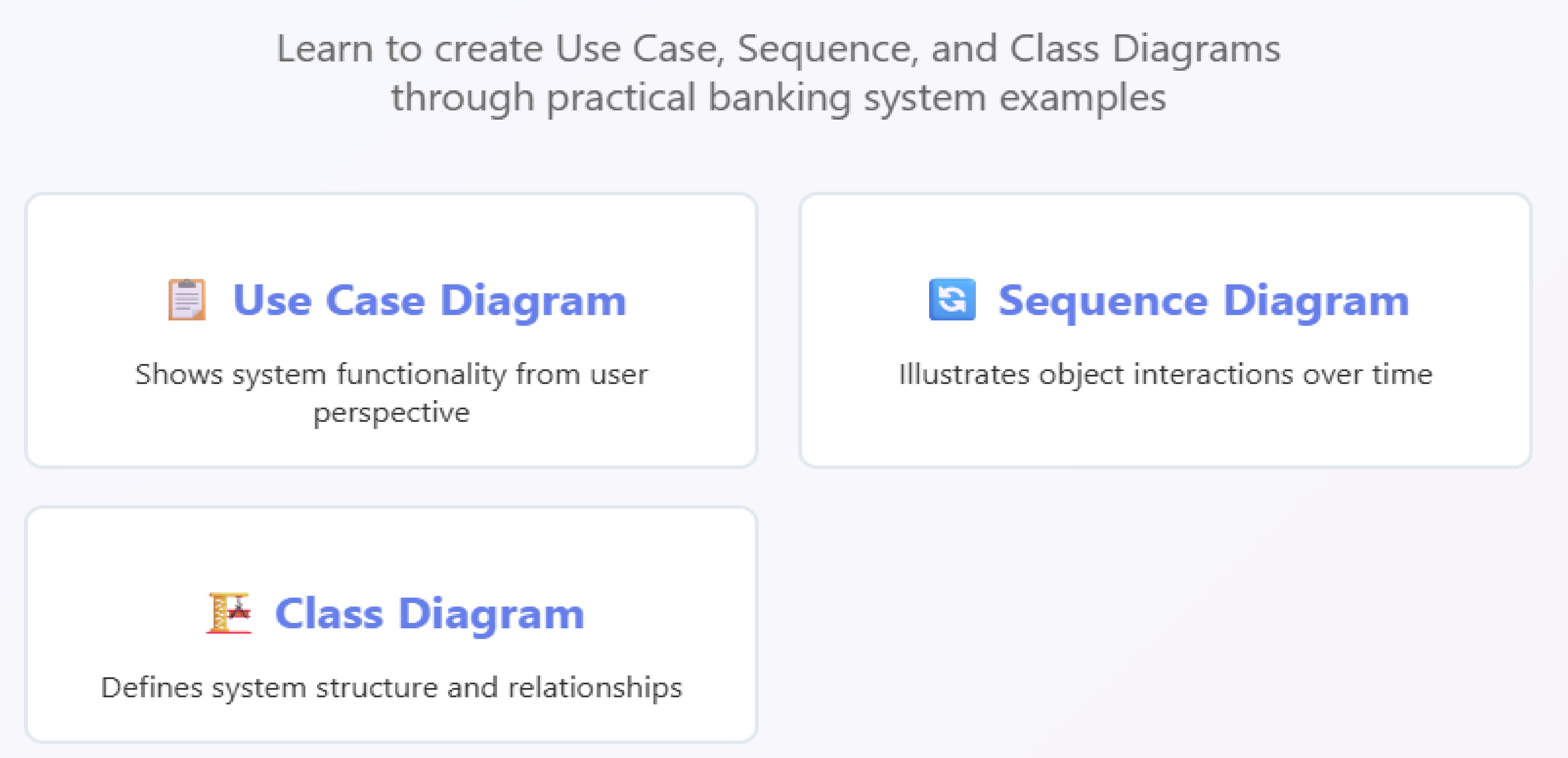
UML Diagrams Training

Creating Professional System Designs



Learn to create Use Case, Sequence, and Class Diagrams through practical banking system examples

📋 Use Case Diagram

Shows system functionality from user perspective

🔄 Sequence Diagram

Illustrates object interactions over time

🏗️ Class Diagram

Defines system structure and relationships

Learning Objectives

By the end of this session, you will be able to:

- ✅ Create comprehensive Use Case diagrams showing actors and system boundaries
- ✅ Design Sequence diagrams that capture object interactions and message flows
- ✅ Develop Class diagrams with proper relationships and attributes
- ✅ Apply UML best practices for clear and professional diagrams
- ✅ Analyze requirements and translate them into visual system designs

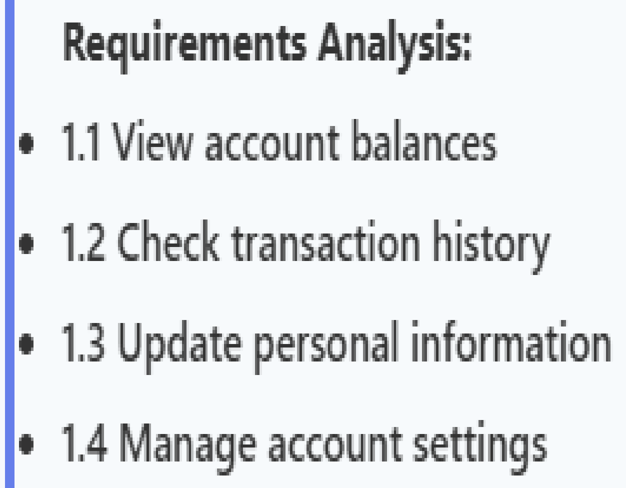
Today's Approach

Step 1: Learn with guided example (Account Management)

Step 2: Practice independently (Transactions)

Step 3: Review and discuss solutions

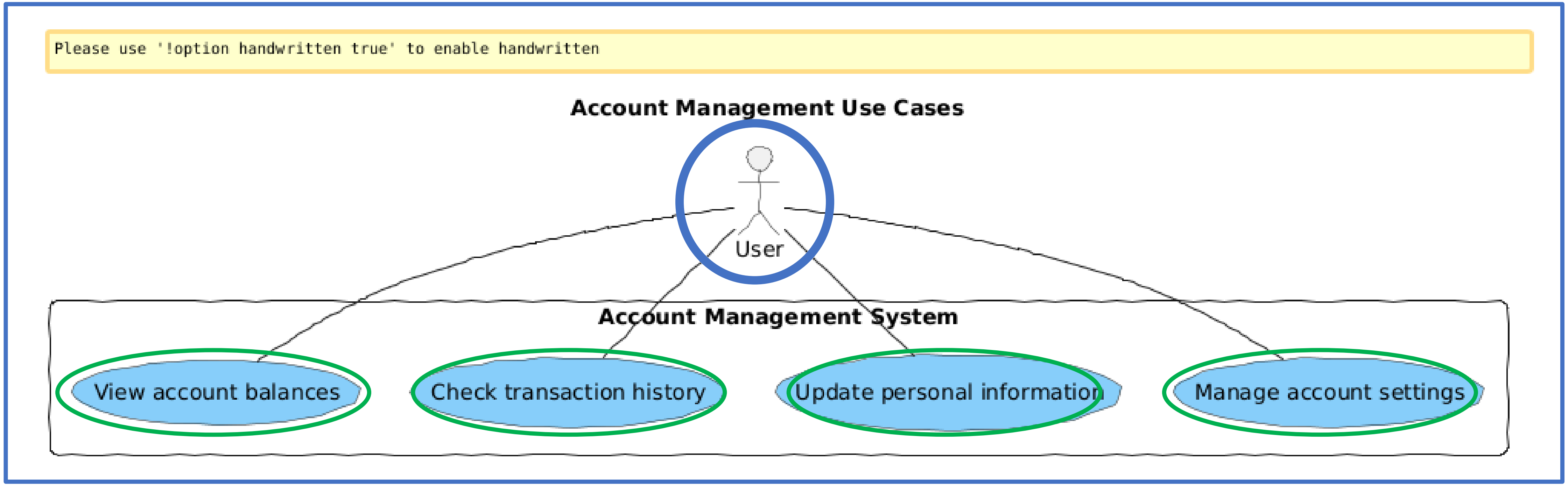
Use Case Diagram - Account Management



Requirements Analysis:

- 1.1 View account balances
- 1.2 Check transaction history
- 1.3 Update personal information
- 1.4 Manage account settings

Use Case Diagram



```
Please use '!option handwritten true' to enable handwritten
```

Account Management Use Cases
User
Account Management System
View account balances
Check transaction history
Update personal information
Manage account settings

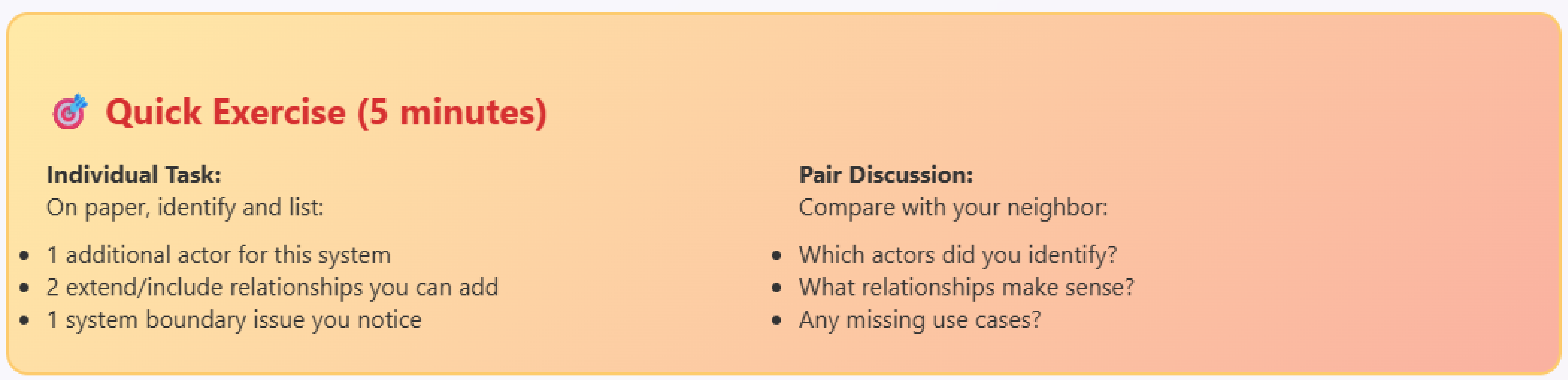
Key Components:

Actor: Customer (external entity)

Use Cases: System functionalities

System Boundary: Defines scope of the banking system

🎯 Classroom Assignment



🎯 Quick Exercise (5 minutes)

Individual Task:
On paper, identify and list:

- 1 additional actor for this system
- 2 extend/include relationships you can add
- 1 system boundary issue you notice

Pair Discussion:
Compare with your neighbor:

- Which actors did you identify?
- What relationships make sense?
- Any missing use cases?

✅ Use Case Exercise - Solutions

Additional Actors Identified:

- Bank Administrator - Manages system settings
- Audit System - Logs all transactions
- Notification Service - Sends alerts

System Boundary Issues:

Issue: AccountSettings appears disconnected
Solution: Should be associated with Account class or integrated within "Manage Account Settings" use case

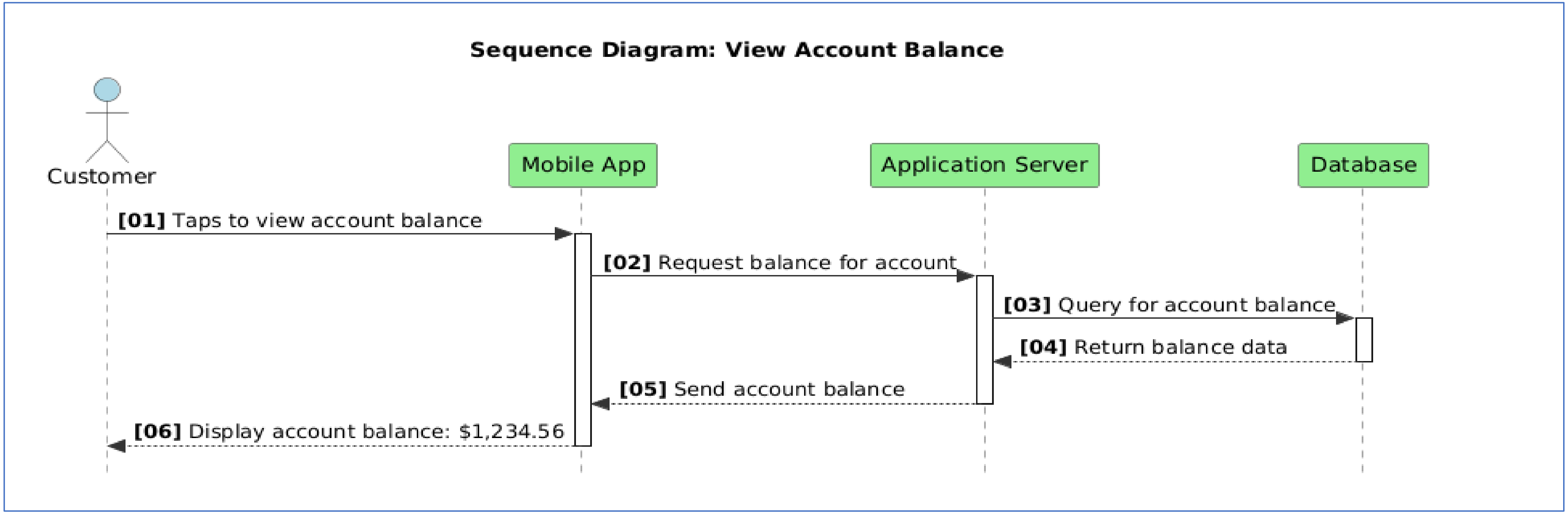
Extend/Include Relationships:

```
View Account Balance
├── «include» → Authenticate User
└── «extend» → Show Account Details

Update Personal Information
├── «include» → Authenticate User
├── «include» → Validate Data
└── «extend» → Send Confirmation Email

Manage Account Settings
├── «include» → Authenticate User
└── «extend» → Update Security Questions
```

💡 **Key Insight:** «include» = mandatory steps, «extend» = optional features

Sequence Diagram - View Account Balance

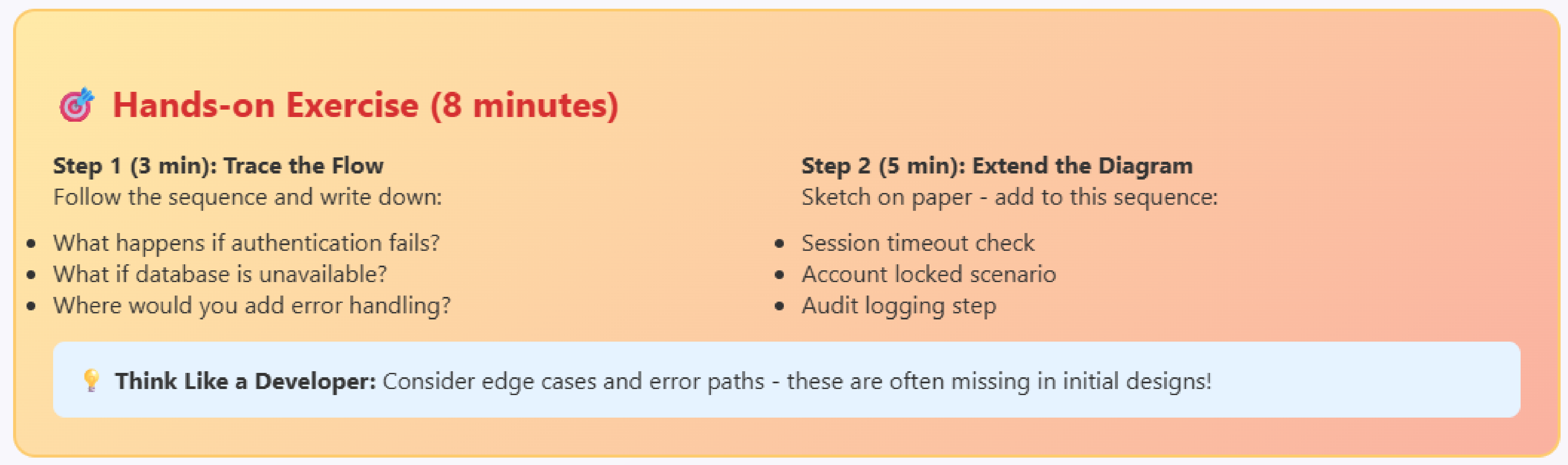
Scenario: Customer views their account balance


Sequence Diagram: View Account Balance
Customer
Mobile App
Application Server
Database
[01] Taps to view account balance
[02] Request balance for account
[03] Query for account balance
[04] Return balance data
[05] Send account balance
[06] Display account balance: $1,234.56

Sequence Diagram Elements:

Actors/Objects: Customer, Web Interface, Account Controller, Database
Messages: Arrows showing communication flow
Lifelines: Vertical dashed lines showing object existence
Time Flow: Top to bottom chronological order

🎯 Classroom Assignment



🎯 Hands-on Exercise (8 minutes)

Step 1 (3 min): Trace the Flow
Follow the sequence and write down:

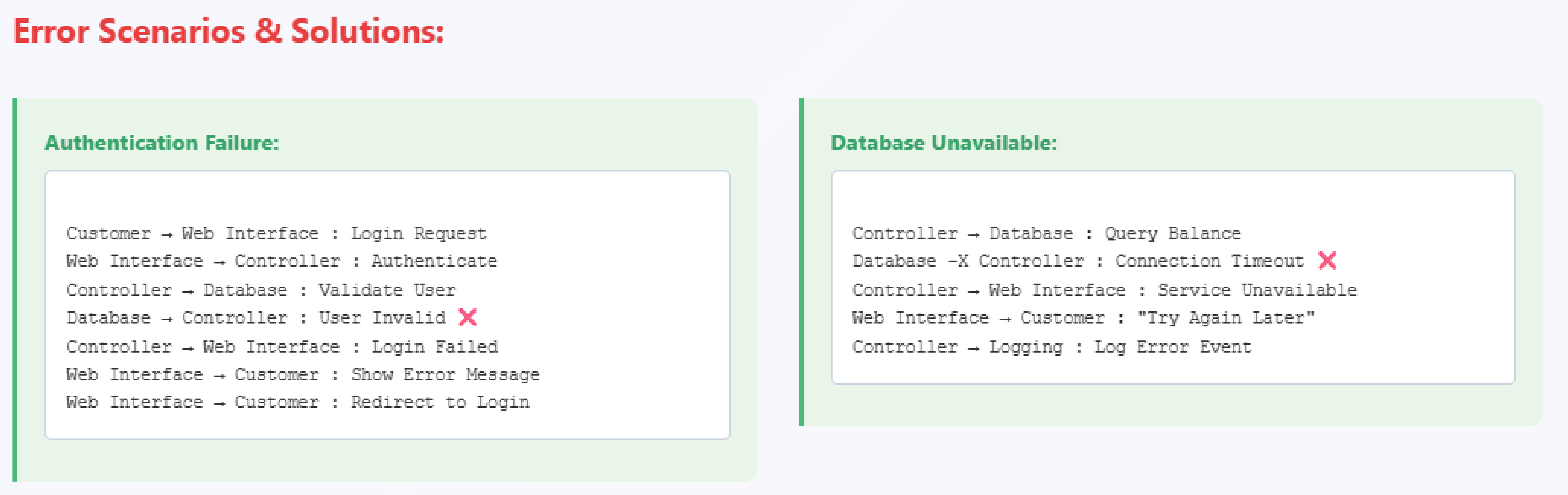
- What happens if authentication fails?
- What if database is unavailable?
- Where would you add error handling?

Step 2 (5 min): Extend the Diagram
Sketch on paper - add to this sequence:

- Session timeout check
- Account locked scenario
- Audit logging step

💡 **Think Like a Developer:** Consider edge cases and error paths - these are often missing in initial designs!

✅ Sequence Exercise - Solutions



Error Scenarios & Solutions:

Authentication Failure:

```
Customer → Web Interface : Login Request
Web Interface → Controller : Authenticate
Controller → Database : Validate User
Database → Controller : User Invalid ❌
Controller → Web Interface : Login Failed
Web Interface → Customer : Show Error Message
Web Interface → Customer : Redirect to Login
```

Database Unavailable:

```
Controller → Database : Query Balance
Database -X Controller : Connection Timeout ❌
Controller → Web Interface : Service Unavailable
Web Interface → Customer : "Try Again Later"
Controller → Logging : Log Error Event
```

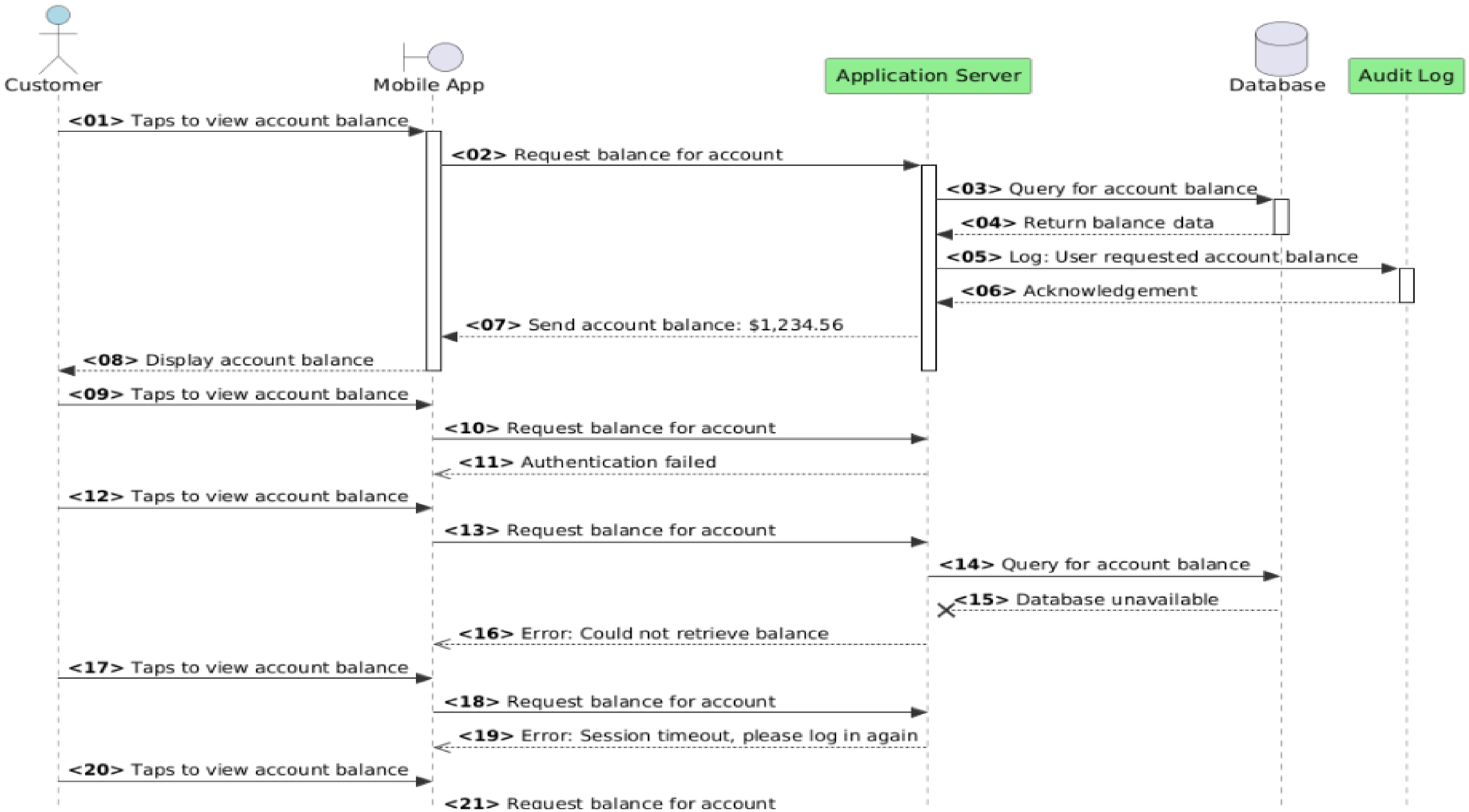
✅ Sequence Exercise - Solutions

Extended Sequence with Enhancements:

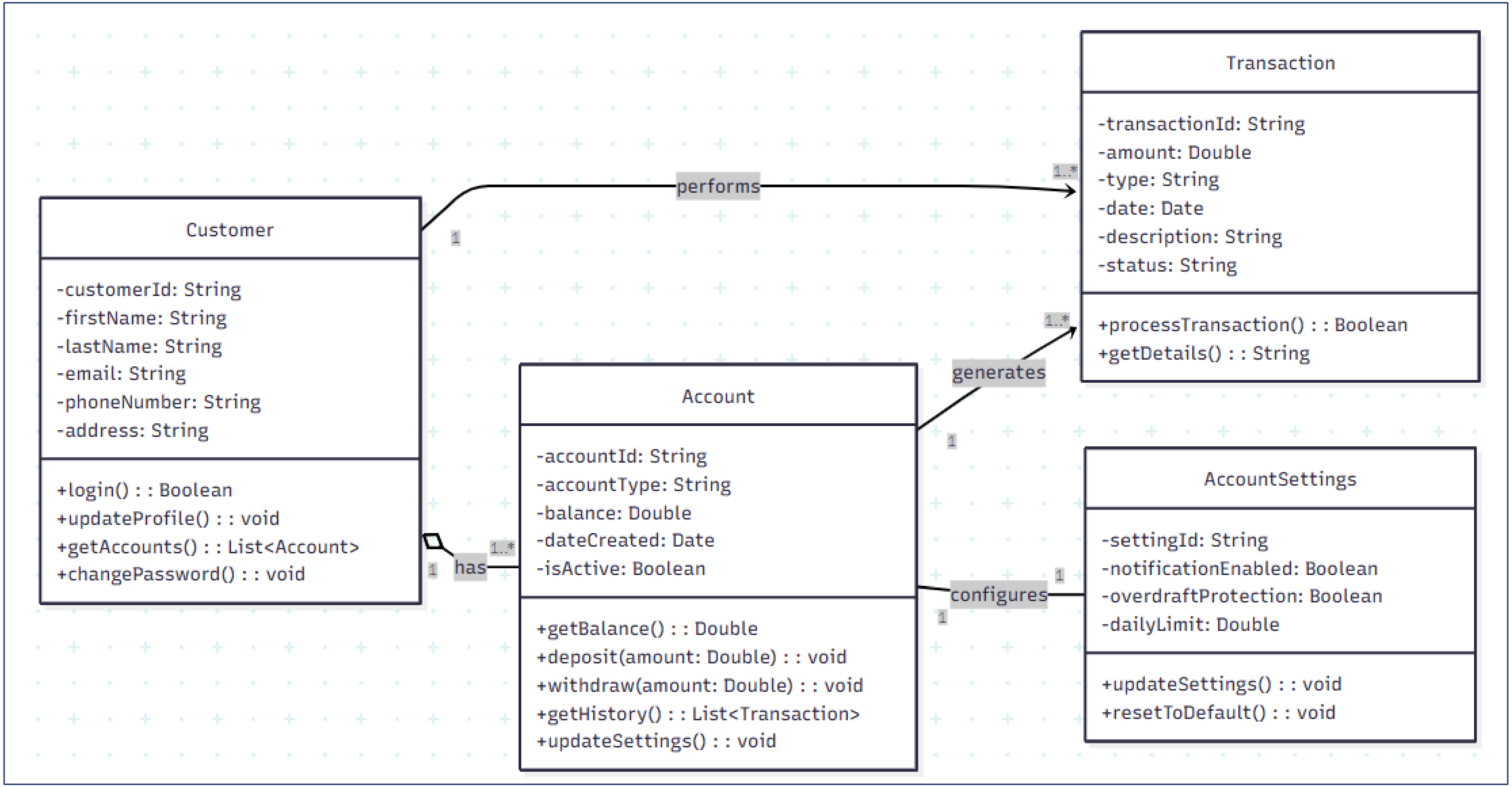
```
Customer → Web Interface : Login Request
Web Interface → SessionManager : Check Session Timeout ✨
SessionManager → Web Interface : Session Valid
Web Interface → Controller : Authenticate
Controller → Database : Validate User
alt [Account Locked] ✨
Database → Controller : Account Locked Status
Controller → Web Interface : Account Locked Error
Web Interface → Customer : Contact Support Message
else [Account Active]
Database → Controller : User Valid
Controller → AuditLogger : Log Login Success ✨
Controller → Web Interface : Login Success
end
Web Interface → Customer : Show Dashboard
```

🎯 **Best Practice:** Always plan for failure scenarios - they're 50% of real-world system behavior!

✅ Sequence Exercise - Solutions


Customer
Mobile App
Application Server
Database
Audit Log
<01> Taps to view account balance
<02> Request balance for account
<03> Query for account balance
<04> Return balance data
<05> Log: User requested account balance
<06> Acknowledgement
<07> Send account balance: $1,234.56
<08> Display account balance
<09> Taps to view account balance
<10> Request balance for account
<11> Authentication failed
<12> Taps to view account balance
<13> Request balance for account
<14> Query for account balance
<15> Database unavailable
<16> Error: Could not retrieve balance
<17> Taps to view account balance
<18> Request balance for account
<19> Error: Session timeout, please log in again
<20> Taps to view account balance
<21> Request balance for account

Class Diagram - Account Management System



Class Diagram - Account Management System

Class Diagram Relationships:

Composition (◆): Customer owns Accounts (strong relationship)
Association (→): Customer has Transactions
Multiplicity: 1..* (one to many relationships)

🎯 Design Challenge (10 minutes)

Individual Task (5 min):
Design on paper:

- Add inheritance to Account class (Savings, Checking)
- Create a SecurityQuestion class
- Define relationships between all classes

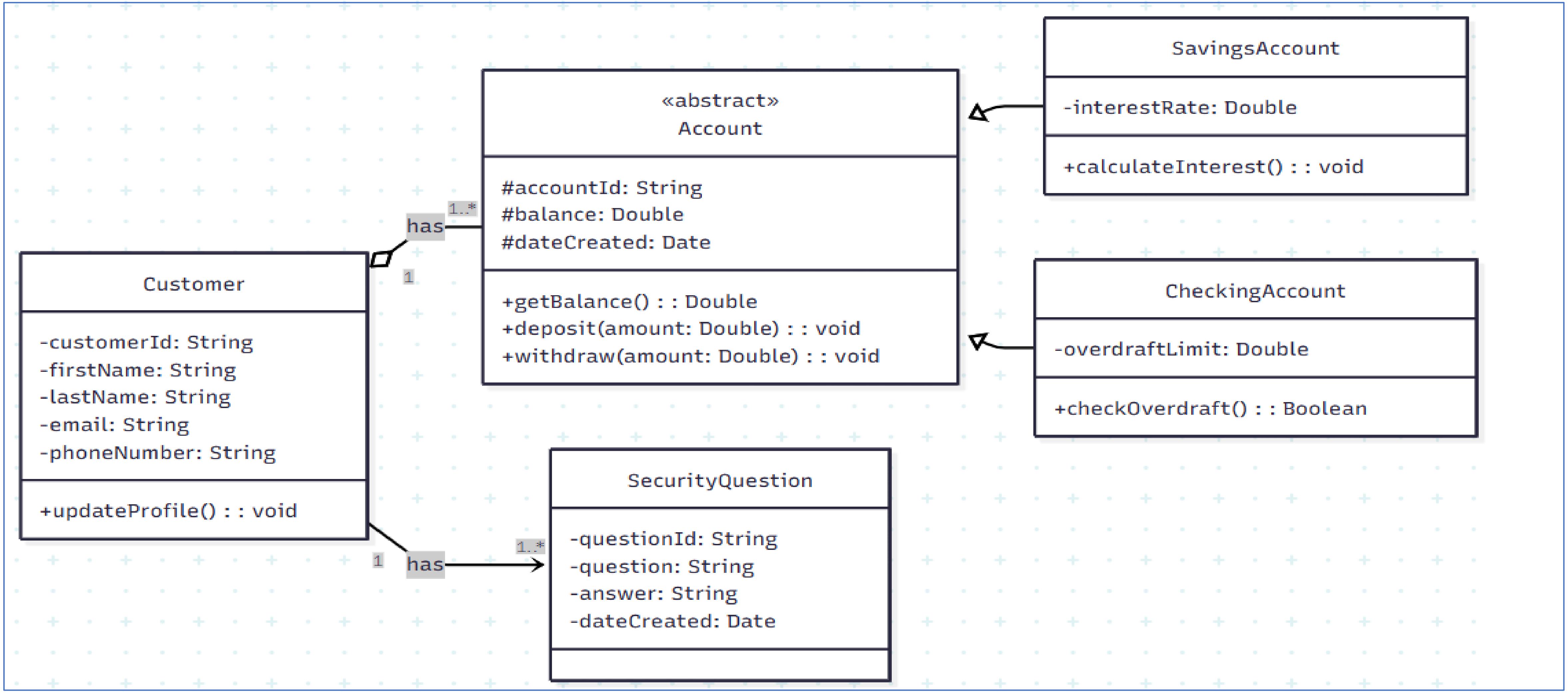
Code Thinking (5 min):
Based on your diagram, write:

- Class declaration for SavingsAccount
- Constructor with proper inheritance
- One unique method for each account type

💻 **Programming Connection:** Notice how UML directly translates to code structure in C++, Python, C#, or Java!

✅ Class Exercise - Solutions

Enhanced Class Diagram with Inheritance:



SavingsAccount

-interestRate: Double

+calculateInterest() : : void

«abstract»
Account

#accountId: String
#balance: Double
#dateCreated: Date

+getBalance() : : Double
+deposit(amount: Double) : : void
+withdraw(amount: Double) : : void

CheckingAccount

-overdraftLimit: Double

+checkOverdraft() : : Boolean

Customer

-customerId: String
-firstName: String
-lastName: String
-email: String
-phoneNumber: String

+updateProfile() : : void

SecurityQuestion

-questionId: String
-question: String
-answer: String
-dateCreated: Date

✅ Class Exercise - Solutions

Code Implementation Examples:

Java/C# Style:

```
public class SavingsAccount extends Account {
private double interestRate;

public SavingsAccount(String accountId,
double initialBalance,
double rate) {
super(accountId, initialBalance);
this.interestRate = rate;
}

public double calculateInterest() {
return getBalance() * interestRate;
}
}
```

Python Style:

```
class CheckingAccount(Account):
def __init__(self, account_id, initial_balance,
overdraft_limit):
super().__init__(account_id, initial_balance)
self.overdraft_limit = overdraft_limit

def check_overdraft(self, amount):
available = self.balance + self.overdraft_limit
return amount <= available

def withdraw(self, amount):
if self.check_overdraft(amount):
super().withdraw(amount)
else:
raise InsufficientFundsError()
```

🏗️ **Architecture Insight:** Notice how inheritance in UML directly maps to object-oriented programming concepts across all languages!

UML Best Practices & Guidelines

✅ Do's

- Use clear, descriptive names
- Keep diagrams simple and focused
- Show only relevant details
- Use consistent notation
- Include multiplicity in relationships
- Validate with stakeholders

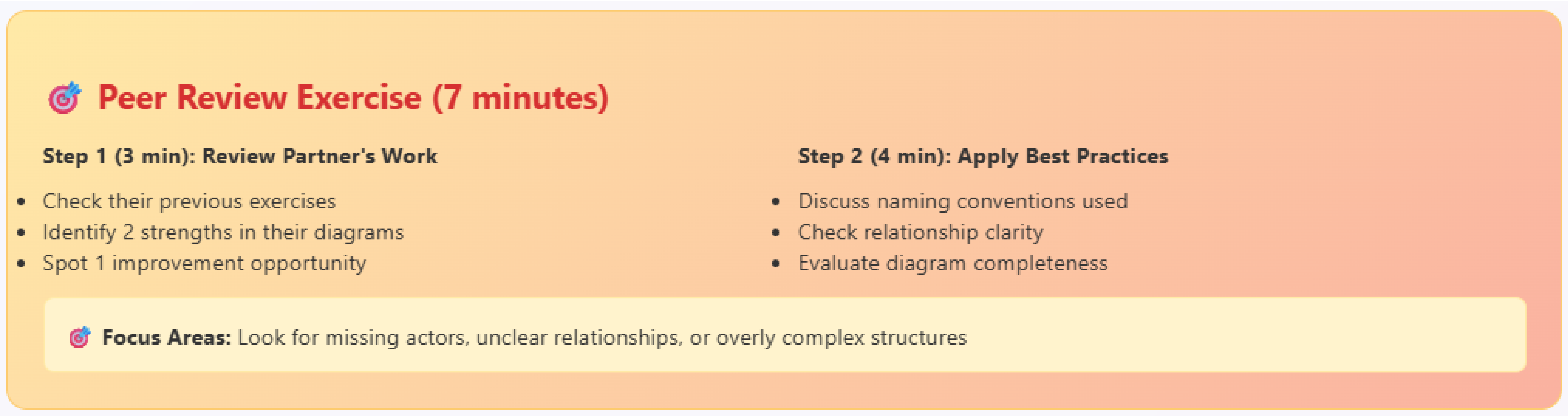
❌ Don'ts

- Don't overcomplicate diagrams
- Avoid crossing lines when possible
- Don't mix abstraction levels
- Don't ignore naming conventions
- Don't create diagrams without purpose
- Don't skip documentation

Diagram Selection Guide:

Use Case Diagrams: For requirements gathering and system scope definition
Sequence Diagrams: For detailed interaction flows and API design
Class Diagrams: For system architecture and database design

Classroom Assignment



Peer Review Exercise (7 minutes)

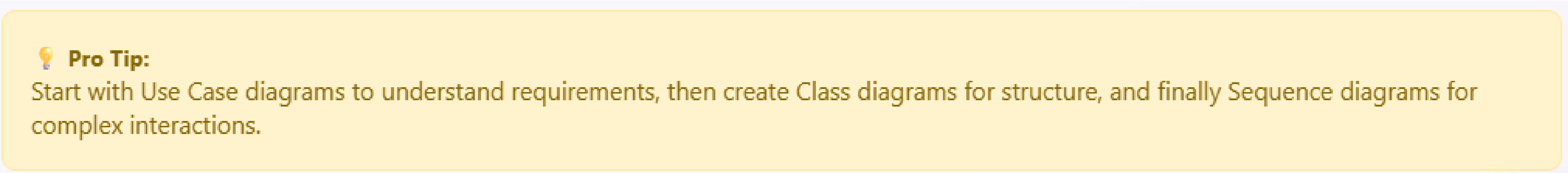
Step 1 (3 min): Review Partner's Work

- Check their previous exercises
- Identify 2 strengths in their diagrams
- Spot 1 improvement opportunity

Step 2 (4 min): Apply Best Practices

- Discuss naming conventions used
- Check relationship clarity
- Evaluate diagram completeness

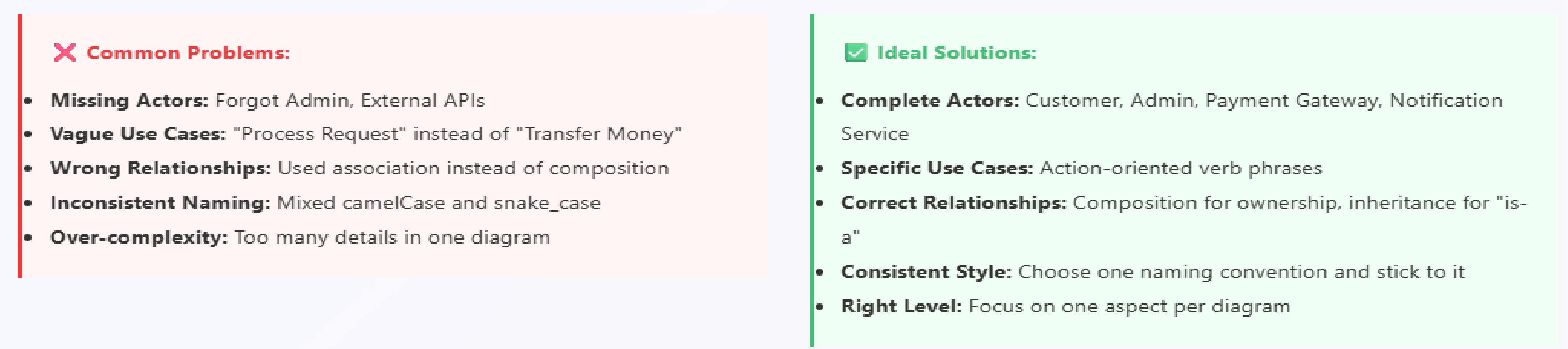
Focus Areas: Look for missing actors, unclear relationships, or overly complex structures



Pro Tip:
Start with Use Case diagrams to understand requirements, then create Class diagrams for structure, and finally Sequence diagrams for complex interactions.

✅ Peer Review Exercise - Solutions

Common Issues Found & Solutions:



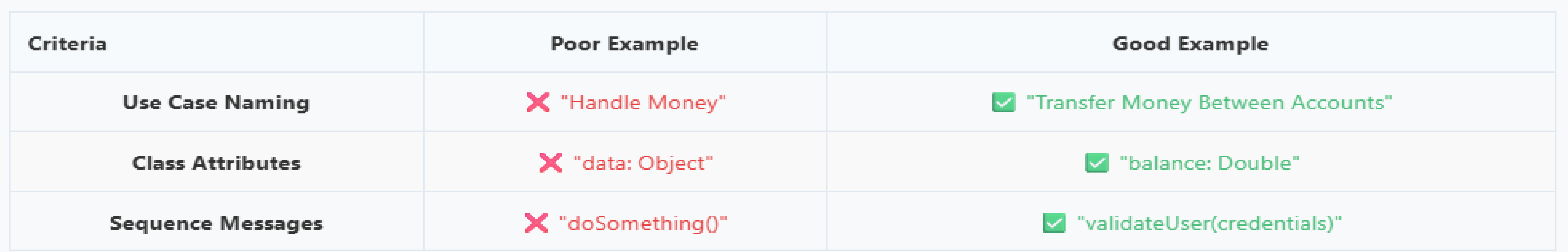
❌ **Common Problems:**

- **Missing Actors:** Forgot Admin, External APIs
- **Vague Use Cases:** "Process Request" instead of "Transfer Money"
- **Wrong Relationships:** Used association instead of composition
- **Inconsistent Naming:** Mixed camelCase and snake_case
- **Over-complexity:** Too many details in one diagram

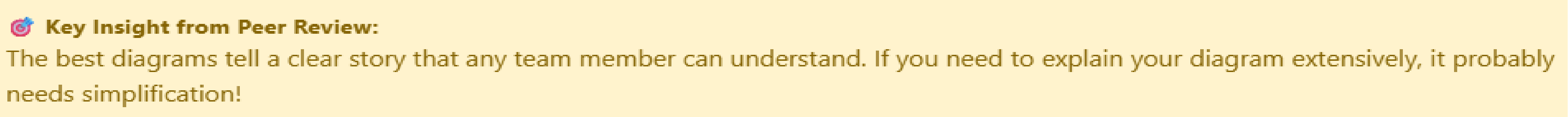
✅ **Ideal Solutions:**

- **Complete Actors:** Customer, Admin, Payment Gateway, Notification Service
- **Specific Use Cases:** Action-oriented verb phrases
- **Correct Relationships:** Composition for ownership, inheritance for "is-a"
- **Consistent Style:** Choose one naming convention and stick to it
- **Right Level:** Focus on one aspect per diagram

Best Practice Checklist Applied:



Criteria	Poor Example	Good Example
Use Case Naming	❌ "Handle Money"	✅ "Transfer Money Between Accounts"
Class Attributes	❌ "data: Object"	✅ "balance: Double"
Sequence Messages	❌ "doSomething()"	✅ "validateUser(credentials)"



🎯 **Key Insight from Peer Review:**

The best diagrams tell a clear story that any team member can understand. If you need to explain your diagram extensively, it probably needs simplification!

🎯 Classroom Exercise

Your Turn: Create UML Diagrams for Transactions

Requirements - Transactions Module:

- 2.1 Transfer money between accounts
- 2.2 Pay bills online
- 2.3 Send money to other users
- 2.4 Schedule recurring payments

Assignment Tasks:

1. Use Case Diagram
• Identify actors (Customer, Bank System, External Payment Gateway)
• Define use cases from the requirements
• Show system boundaries

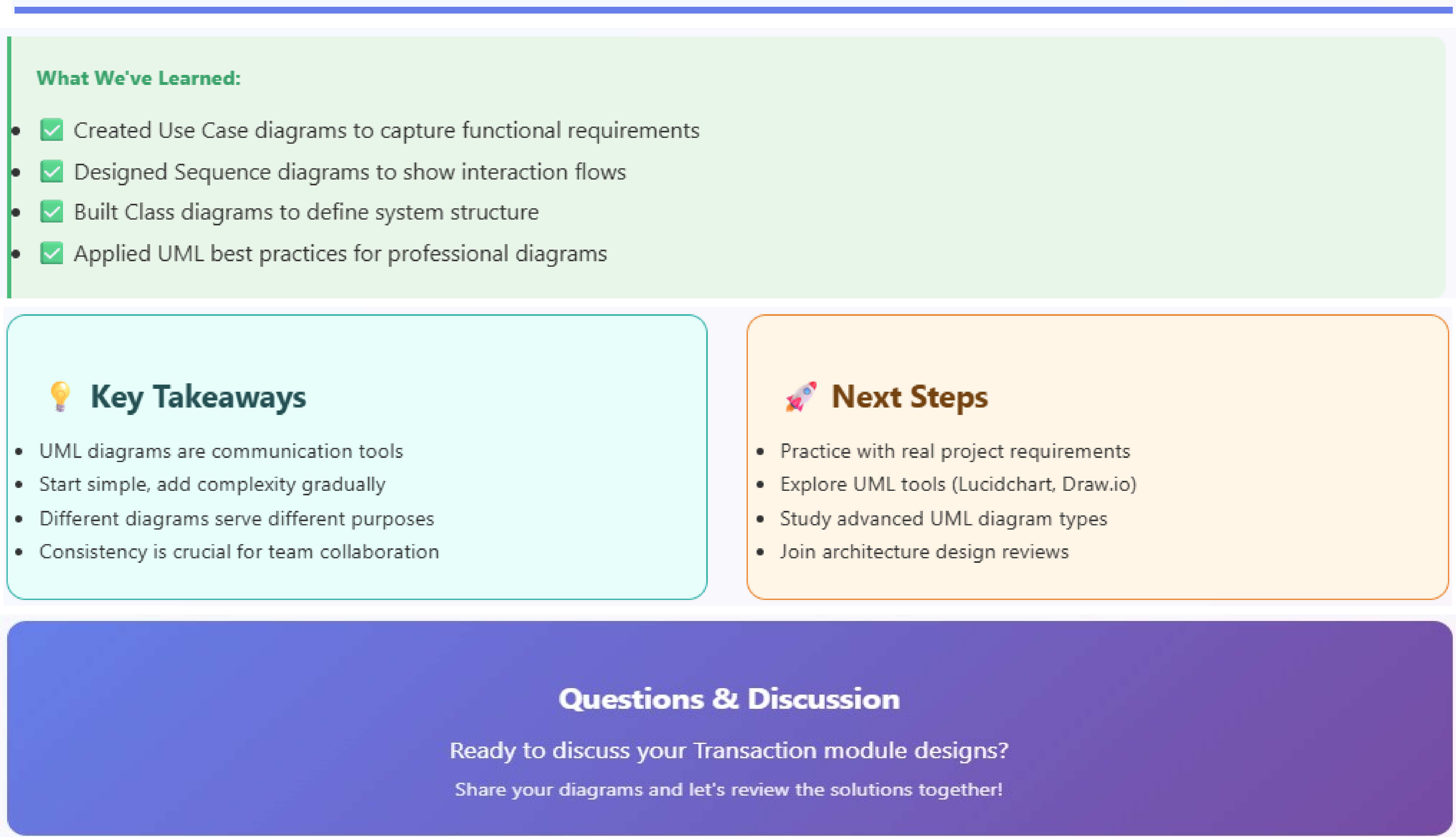
2. Sequence Diagram
• Choose: "Transfer money between accounts"
• Show: Customer, UI, TransferController, Account, Database
• Include: Validation, balance checks, transaction processing

3. Class Diagram
• Design: Transaction, Transfer, BillPayment, RecurringPayment classes
• Show: Attributes, methods, relationships
• Consider: Inheritance and composition

⏰ Time Allocation: 45 minutes total (15 minutes per diagram)

Apply all the techniques and best practices from the previous exercises!

Summary & Next Steps



What We've Learned:

- ✅ Created Use Case diagrams to capture functional requirements
- ✅ Designed Sequence diagrams to show interaction flows
- ✅ Built Class diagrams to define system structure
- ✅ Applied UML best practices for professional diagrams

💡 Key Takeaways

- UML diagrams are communication tools
- Start simple, add complexity gradually
- Different diagrams serve different purposes
- Consistency is crucial for team collaboration

🚀 Next Steps

- Practice with real project requirements
- Explore UML tools (Lucidchart, Draw.io)
- Study advanced UML diagram types
- Join architecture design reviews

Questions & Discussion

Ready to discuss your Transaction module designs?

Share your diagrams and let's review the solutions together!

ER Diagrams Training

ER Diagrams Training

Database Design & Data Modelling

Master Entity-Relationship diagrams for robust database design through practical banking system examples

🏗️ Entities

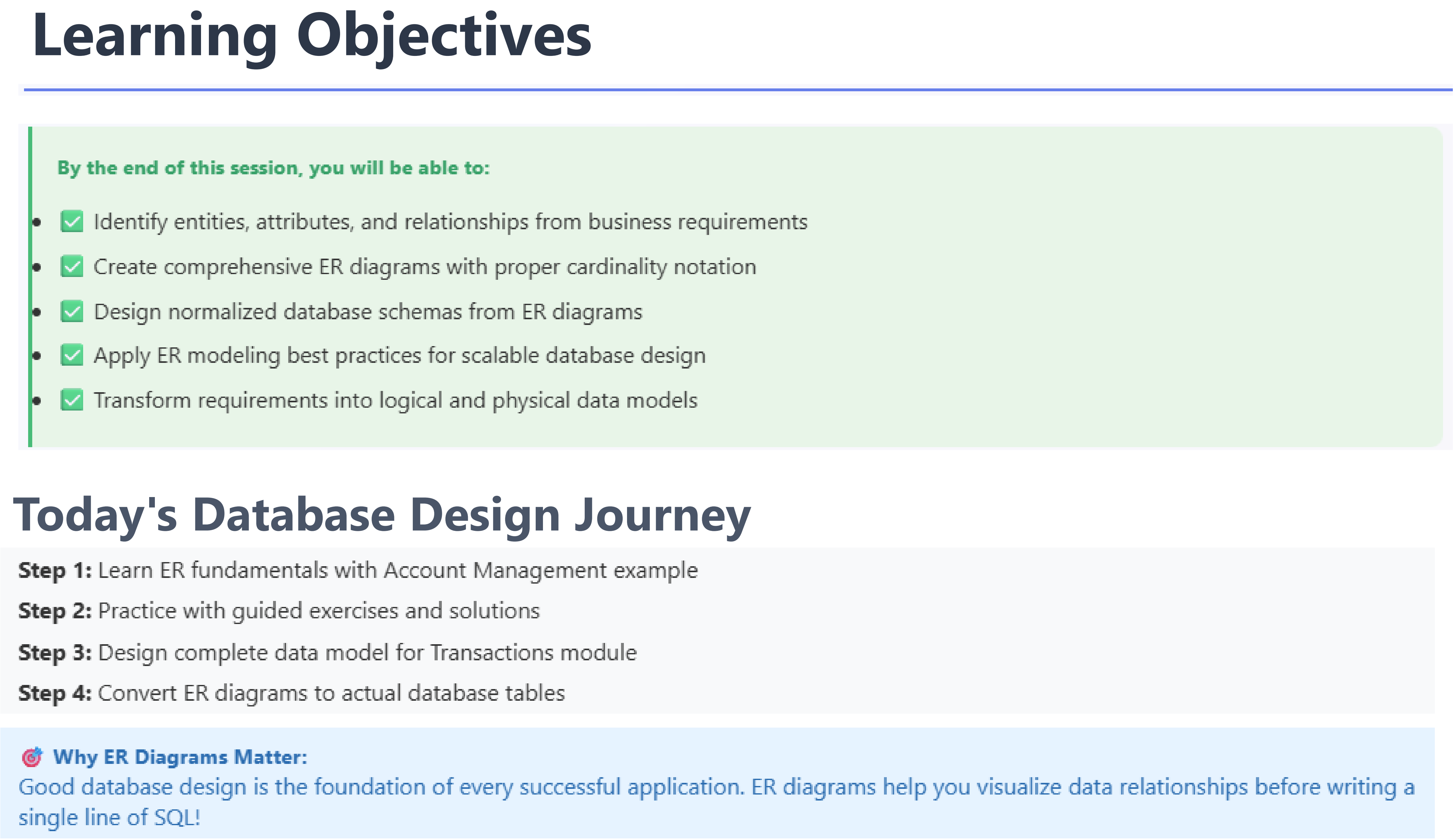
Real-world objects that store data

🔗 Relationships

Connections between entities

📋 Attributes

Properties that describe entities



Learning Objectives

By the end of this session, you will be able to:

- ✅ Identify entities, attributes, and relationships from business requirements
- ✅ Create comprehensive ER diagrams with proper cardinality notation
- ✅ Design normalized database schemas from ER diagrams
- ✅ Apply ER modeling best practices for scalable database design
- ✅ Transform requirements into logical and physical data models

Today's Database Design Journey

Step 1: Learn ER fundamentals with Account Management example

Step 2: Practice with guided exercises and solutions

Step 3: Design complete data model for Transactions module

Step 4: Convert ER diagrams to actual database tables

🎯 **Why ER Diagrams Matter:**
Good database design is the foundation of every successful application. ER diagrams help you visualize data relationships before writing a single line of SQL!

ER Diagram Fundamentals

Entities

Customer

Account

Transaction

Nouns that represent real-world objects or concepts

Relationships

owns

performs

belongs_to

Verbs that describe associations between entities

Attributes

customer_id

first_name

balance

Properties that describe entities

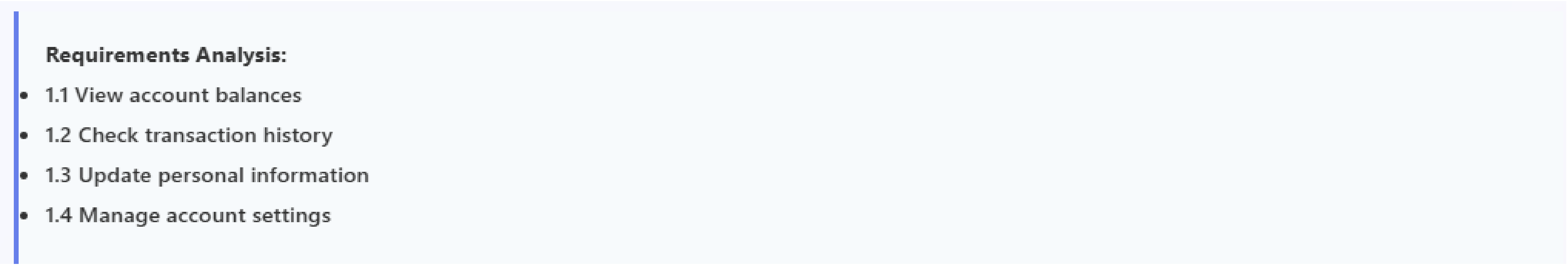
Cardinality Notation:

One-to-One (1:1)
Customer ——— Profile
Each customer has exactly one profile

One-to-Many (1:M)
Customer ——< Account
One customer can have multiple accounts

Many-to-Many (M:N)
Account >——< Service
Accounts can use multiple services

ER Diagram - Account Management System



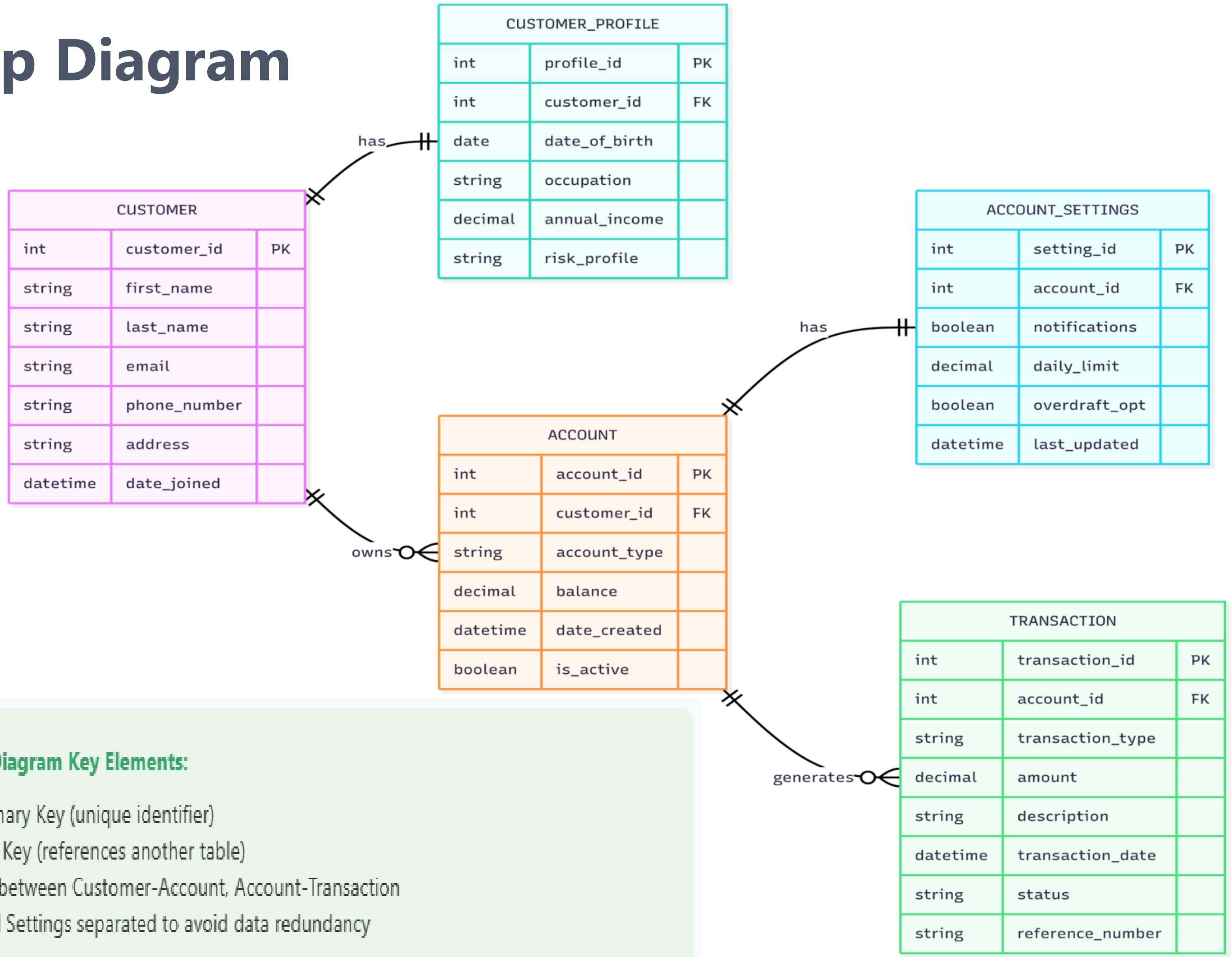
Requirements Analysis:

- 1.1 View account balances
- 1.2 Check transaction history
- 1.3 Update personal information
- 1.4 Manage account settings

Entity-Relationship Diagram

ER Diagram - Account Management System

Entity-Relationship Diagram



CUSTOMER		
int	customer_id	PK
string	first_name	
string	last_name	
string	email	
string	phone_number	
string	address	
datetime	date_joined	

CUSTOMER_PROFILE		
int	profile_id	PK
int	customer_id	FK
date	date_of_birth	
string	occupation	
decimal	annual_income	
string	risk_profile	

ACCOUNT		
int	account_id	PK
int	customer_id	FK
string	account_type	
decimal	balance	
datetime	date_created	
boolean	is_active	

ACCOUNT_SETTINGS		
int	setting_id	PK
int	account_id	FK
boolean	notifications	
decimal	daily_limit	
boolean	overdraft_opt	
datetime	last_updated	

TRANSACTION		
int	transaction_id	PK
int	account_id	FK
string	transaction_type	
decimal	amount	
string	description	
datetime	transaction_date	
string	status	
string	reference_number	

ER Diagram Key Elements:

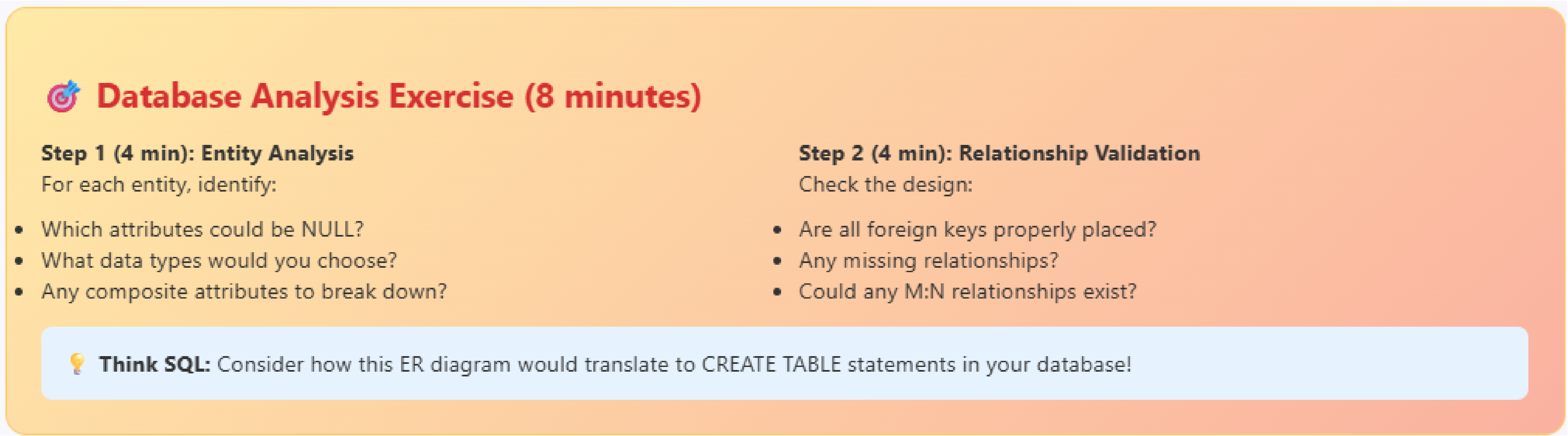
PK: Primary Key (unique identifier)

FK: Foreign Key (references another table)

Cardinality: 1:M relationships between Customer-Account, Account-Transaction

Normalization: Profile and Settings separated to avoid data redundancy

🎯 Classroom Exercise



🎯 Database Analysis Exercise (8 minutes)

Step 1 (4 min): Entity Analysis

For each entity, identify:

- Which attributes could be NULL?
- What data types would you choose?
- Any composite attributes to break down?

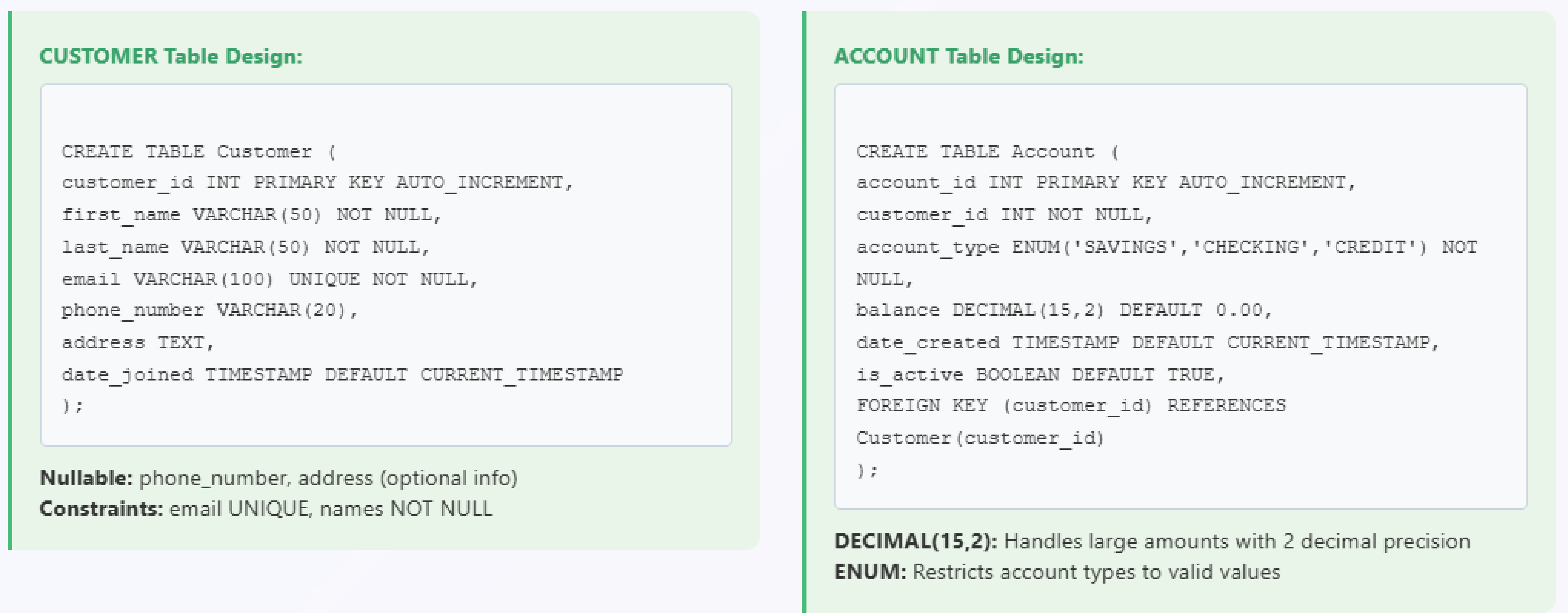
Step 2 (4 min): Relationship Validation

Check the design:

- Are all foreign keys properly placed?
- Any missing relationships?
- Could any M:N relationships exist?

💡 **Think SQL:** Consider how this ER diagram would translate to CREATE TABLE statements in your database!

✅ Database Analysis Exercise - Solutions



Entity Analysis - Data Types & Constraints:

CUSTOMER Table Design:

```
CREATE TABLE Customer (
customer_id INT PRIMARY KEY AUTO_INCREMENT,
first_name VARCHAR(50) NOT NULL,
last_name VARCHAR(50) NOT NULL,
email VARCHAR(100) UNIQUE NOT NULL,
phone_number VARCHAR(20),
address TEXT,
date_joined TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Nullable: phone_number, address (optional info)
Constraints: email UNIQUE, names NOT NULL

ACCOUNT Table Design:

```
CREATE TABLE Account (
account_id INT PRIMARY KEY AUTO_INCREMENT,
customer_id INT NOT NULL,
account_type ENUM('SAVINGS','CHECKING','CREDIT') NOT
NULL,
balance DECIMAL(15,2) DEFAULT 0.00,
date_created TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
is_active BOOLEAN DEFAULT TRUE,
FOREIGN KEY (customer_id) REFERENCES
Customer(customer_id)
);
```

DECIMAL(15,2): Handles large amounts with 2 decimal precision
ENUM: Restricts account types to valid values

✅ Database Analysis Exercise - Solutions

Relationship Validation - Issues & Fixes:

Potential Issue	Analysis	Solution
Composite Address	Should address be broken down?	Yes - street, city, state, zip for better queries
Transaction Categories	Missing transaction categorization	Add category_id FK to Transaction table
Account Joint Ownership	Current design: 1 customer per account	Create AccountHolder junction table for M:N

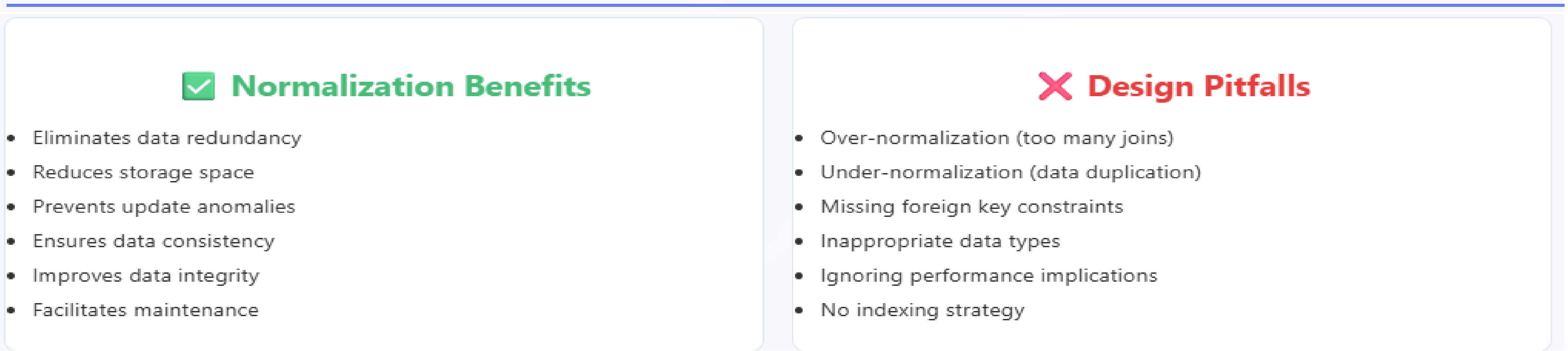
Enhanced ER Model:

Additional Entities Identified:

- TransactionCategory - For expense tracking and reporting
- AccountHolder - Junction table for joint accounts
- Address - Normalized address components

Database Insight: Real banking systems often have 50+ tables - start simple and normalize as requirements grow!

Database Normalisation & ER Best Practices

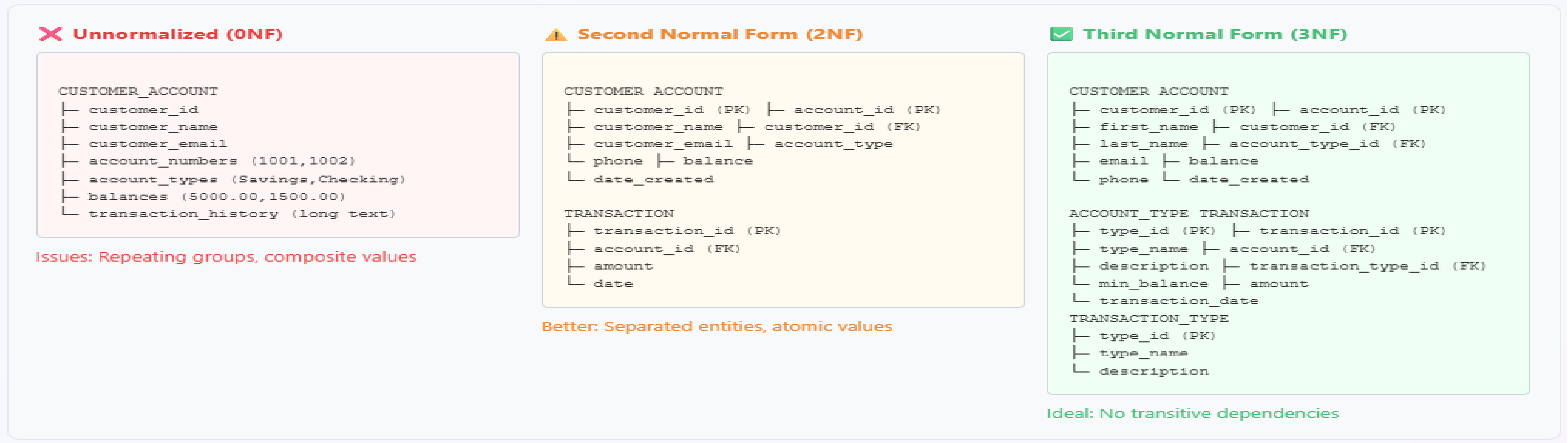


✅ Normalization Benefits

- Eliminates data redundancy
- Reduces storage space
- Prevents update anomalies
- Ensures data consistency
- Improves data integrity
- Facilitates maintenance

❌ Design Pitfalls

- Over-normalization (too many joins)
- Under-normalization (data duplication)
- Missing foreign key constraints
- Inappropriate data types
- Ignoring performance implications
- No indexing strategy



❌ Unnormalized (0NF)

```
CUSTOMER_ACCOUNT
├─ customer_id
├─ customer_name
├─ customer_email
├─ account_numbers (1001,1002)
├─ account_types (Savings,Checking)
├─ balances (5000.00,1500.00)
└─ transaction_history (long text)
```

Issues: Repeating groups, composite values

⚠ Second Normal Form (2NF)

```
CUSTOMER ACCOUNT
├─ customer_id (PK) ├─ account_id (PK)
├─ customer_name ├─ customer_id (FK)
├─ customer_email ├─ account_type
└─ phone ├─ balance
└─ date_created

TRANSACTION
├─ transaction_id (PK)
├─ account_id (FK)
├─ amount
└─ date
```

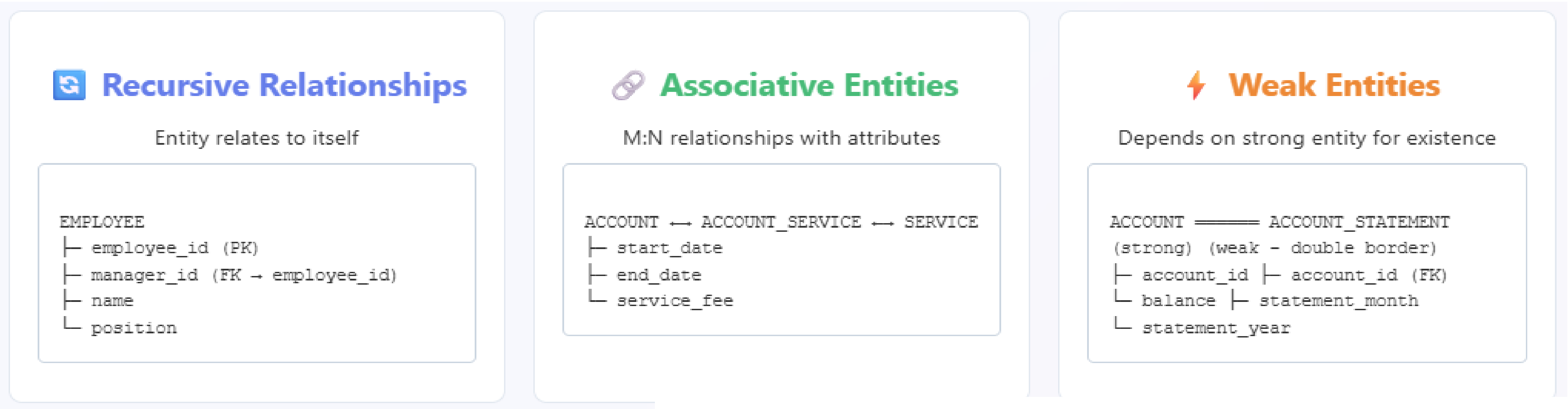
Better: Separated entities, atomic values

✅ Third Normal Form (3NF)

```
CUSTOMER ACCOUNT
├─ customer_id (PK) ├─ account_id (PK)
├─ first_name ├─ customer_id (FK)
├─ last_name ├─ account_type_id (FK)
├─ email ├─ balance
└─ phone └─ date_created

ACCOUNT_TYPE TRANSACTION
├─ type_id (PK) ├─ transaction_id (PK)
├─ type_name ├─ account_id (FK)
├─ description ├─ transaction_type_id (FK)
└─ min_balance ├─ amount
└─ transaction_date
TRANSACTION_TYPE
├─ type_id (PK)
├─ type_name
└─ description
```

Ideal: No transitive dependencies

Advanced ER Modeling Concepts



🔄 Recursive Relationships

Entity relates to itself

```
EMPLOYEE
├─ employee_id (PK)
├─ manager_id (FK → employee_id)
├─ name
└─ position
```

🔗 Associative Entities

M:N relationships with attributes

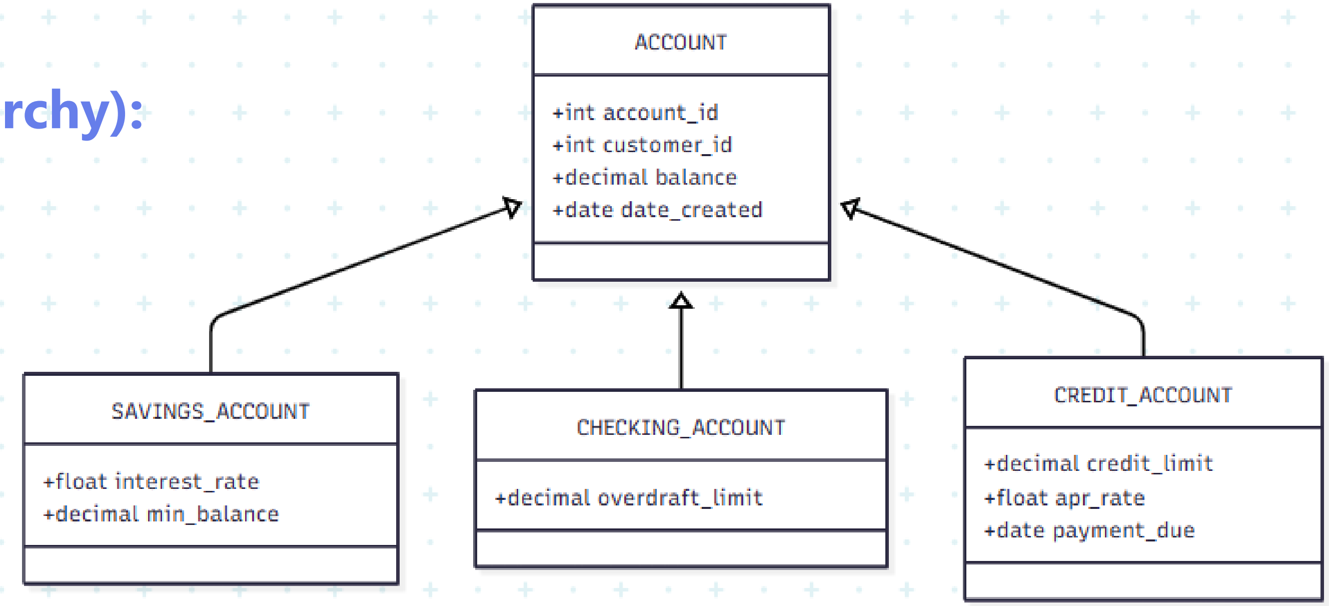
```
ACCOUNT ←→ ACCOUNT_SERVICE ←→ SERVICE
├─ start_date
├─ end_date
└─ service_fee
```

⚡ Weak Entities

Depends on strong entity for existence

```
ACCOUNT ══════ ACCOUNT_STATEMENT
(strong)  (weak - double border)
├─ account_id ├─ account_id (FK)
└─ balance ├─ statement_month
└─ statement_year
```

Inheritance in ER Diagrams (ISA Hierarchy):



ACCOUNT

+int account_id
+int customer_id
+decimal balance
+date date_created

SAVINGS_ACCOUNT

+float interest_rate
+decimal min_balance

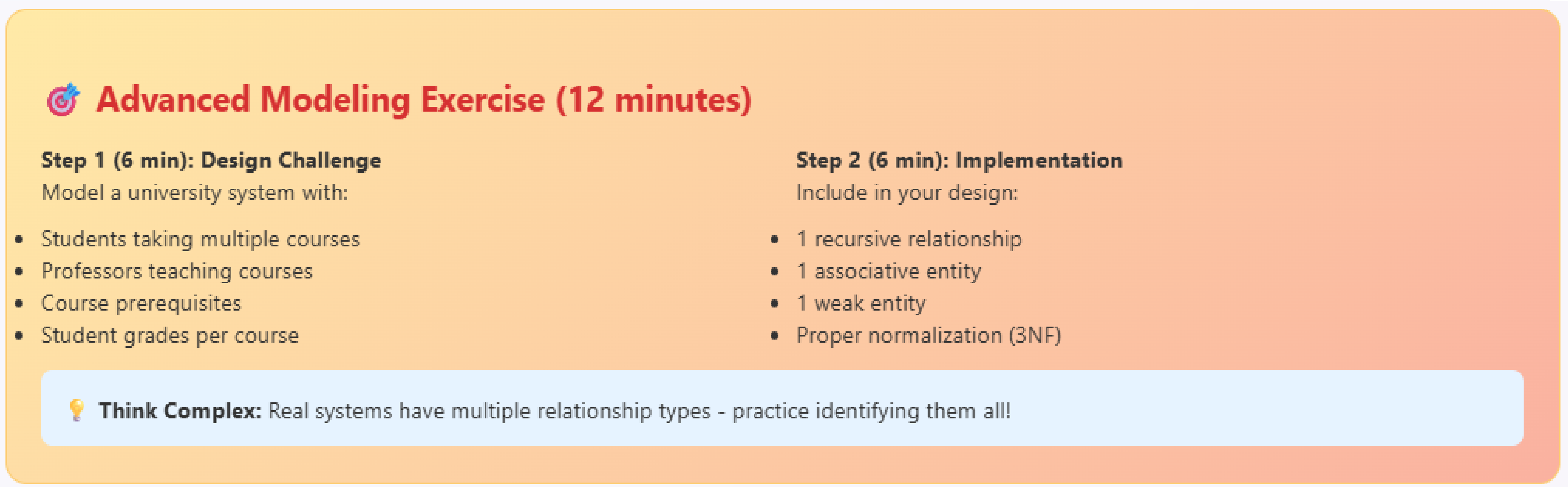
CHECKING_ACCOUNT

+decimal overdraft_limit

CREDIT_ACCOUNT

+decimal credit_limit
+float apr_rate
+date payment_due

Classroom Exercise



Advanced Modeling Exercise (12 minutes)

Step 1 (6 min): Design Challenge

Model a university system with:

- Students taking multiple courses
- Professors teaching courses
- Course prerequisites
- Student grades per course

Step 2 (6 min): Implementation

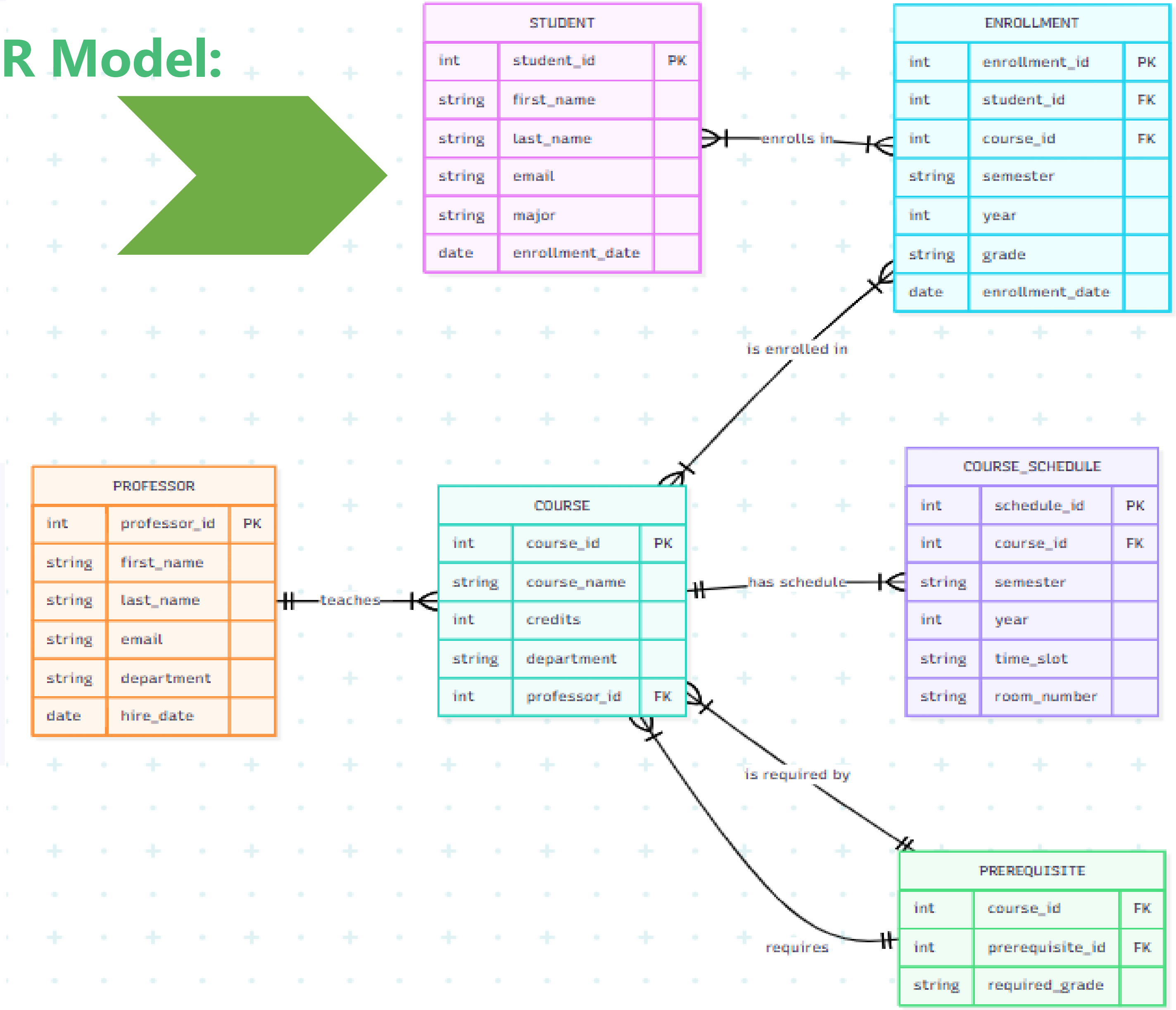
Include in your design:

- 1 recursive relationship
- 1 associative entity
- 1 weak entity
- Proper normalization (3NF)

💡 **Think Complex:** Real systems have multiple relationship types - practice identifying them all!

✅ Advanced Modeling Exercise - Solutions

University System - Complete ER Model:



STUDENT		
int	student_id	PK
string	first_name	
string	last_name	
string	email	
string	major	
date	enrollment_date	

ENROLLMENT		
int	enrollment_id	PK
int	student_id	FK
int	course_id	FK
string	semester	
int	year	
string	grade	
date	enrollment_date	

PROFESSOR		
int	professor_id	PK
string	first_name	
string	last_name	
string	email	
string	department	
date	hire_date	

COURSE		
int	course_id	PK
string	course_name	
int	credits	
string	department	
int	professor_id	FK

COURSE_SCHEDULE		
int	schedule_id	PK
int	course_id	FK
string	semester	
int	year	
string	time_slot	
string	room_number	

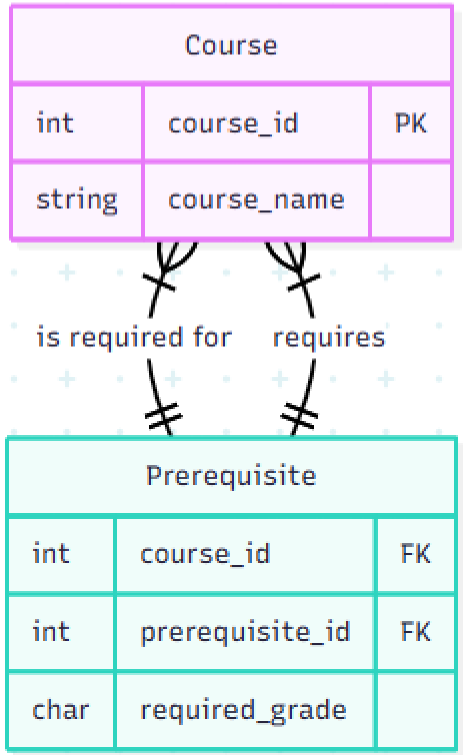
PREREQUISITE		
int	course_id	FK
int	prerequisite_id	FK
string	required_grade	

Advanced Concepts Applied:

- **Recursive Relationship:** PREREQUISITE table (course requires other courses)
- **Associative Entity:** ENROLLMENT (M:N with additional attributes)
- **Weak Entity:** COURSE_SCHEDULE (depends on COURSE for existence)
- **3NF Compliance:** No transitive dependencies

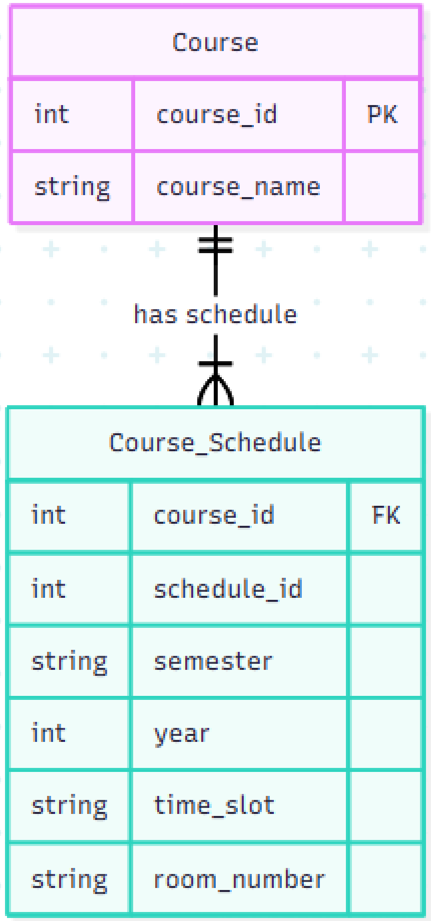
✅ Advanced Modeling Exercise - Solutions

SQL Implementation:



Course		
int	course_id	PK
string	course_name	

Prerequisite		
int	course_id	FK
int	prerequisite_id	FK
char	required_grade	



Course		
int	course_id	PK
string	course_name	

Course_Schedule		
int	course_id	FK
int	schedule_id	
string	semester	
int	year	
string	time_slot	
string	room_number	

```
-- Recursive relationship
CREATE TABLE Prerequisite (
course_id INT,
prerequisite_id INT,
required_grade CHAR(2),
PRIMARY KEY (course_id, prerequisite_id),
FOREIGN KEY (course_id) REFERENCES Course(course_id),
FOREIGN KEY (prerequisite_id) REFERENCES Course(course_id)
);

-- Weak entity with composite key
CREATE TABLE Course_Schedule (
course_id INT,
schedule_id INT,
semester VARCHAR(10),
year INT,
time_slot VARCHAR(20),
room_number VARCHAR(10),
PRIMARY KEY (course_id, schedule_id),
FOREIGN KEY (course_id) REFERENCES Course(course_id)
);
```

🏗️ **Architecture Insight:** Notice how complex business rules are elegantly captured through relationship design and constraints!

Summary & Next Steps

What We've Mastered:

- ✅ Entity identification and attribute design
- ✅ Relationship modeling with proper cardinalities
- ✅ Database normalization to 3NF
- ✅ Advanced concepts: weak entities, inheritance, recursive relationships
- ✅ SQL implementation from ER designs

💡 Key Takeaways

- ER diagrams bridge business requirements and database implementation
- Good normalization prevents future headaches
- Cardinality modeling is critical for data integrity
- Performance considerations guide design decisions

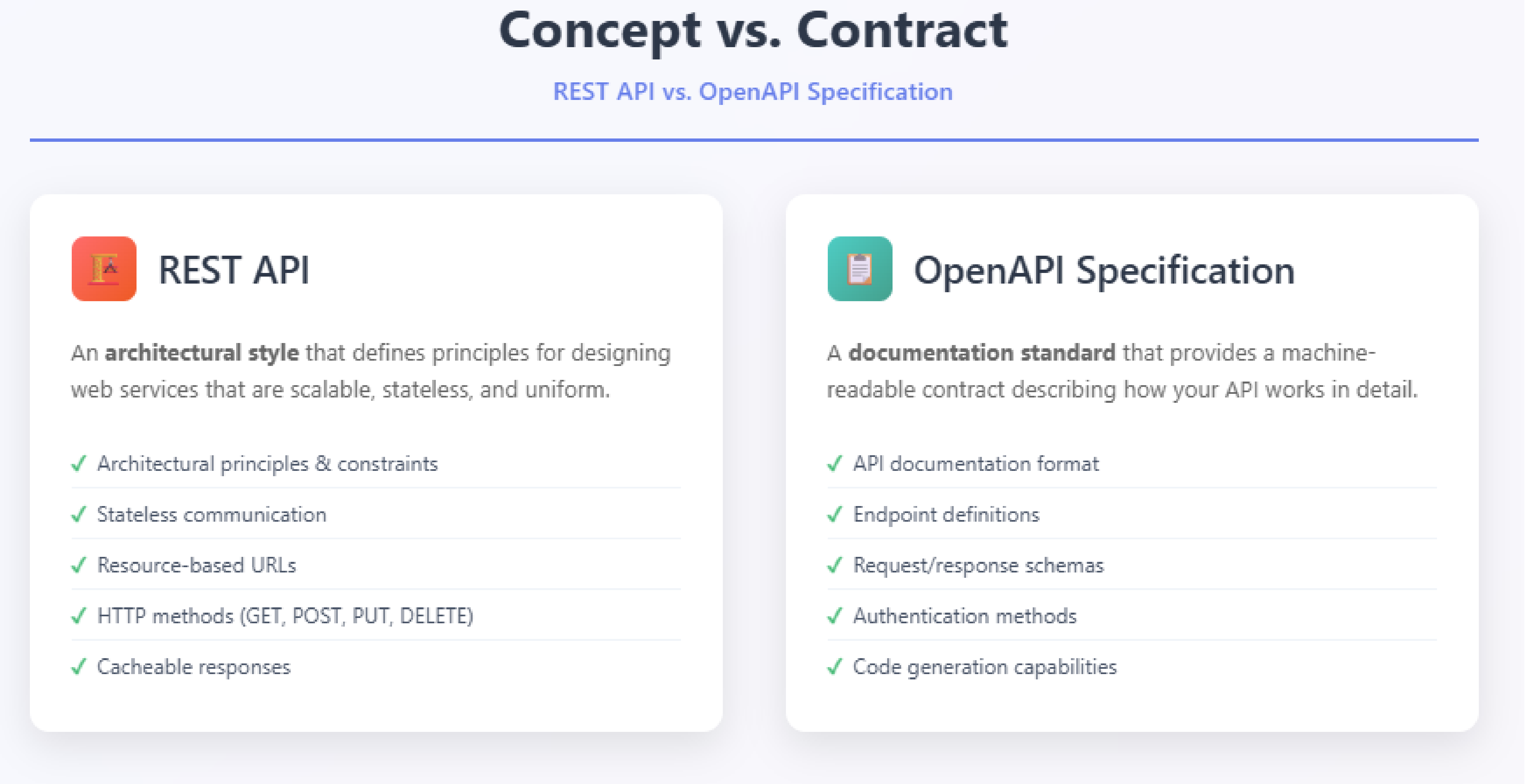
🚀 Next Steps

- Practice with real project requirements
- Explore ER tools (ERDPlus, Lucidchart, Draw.io)
- Study database optimization techniques
- Learn NoSQL data modeling patterns

🗄️ From ER to Code:

Your ER diagrams directly inform ORM configurations in frameworks like Hibernate (Java), Entity Framework (C#), SQLAlchemy (Python), and Sequelize (JavaScript). Master ER modeling, and database programming becomes much more intuitive!

API Specs Planning with Swagger



Concept vs. Contract

REST API vs. OpenAPI Specification

REST API

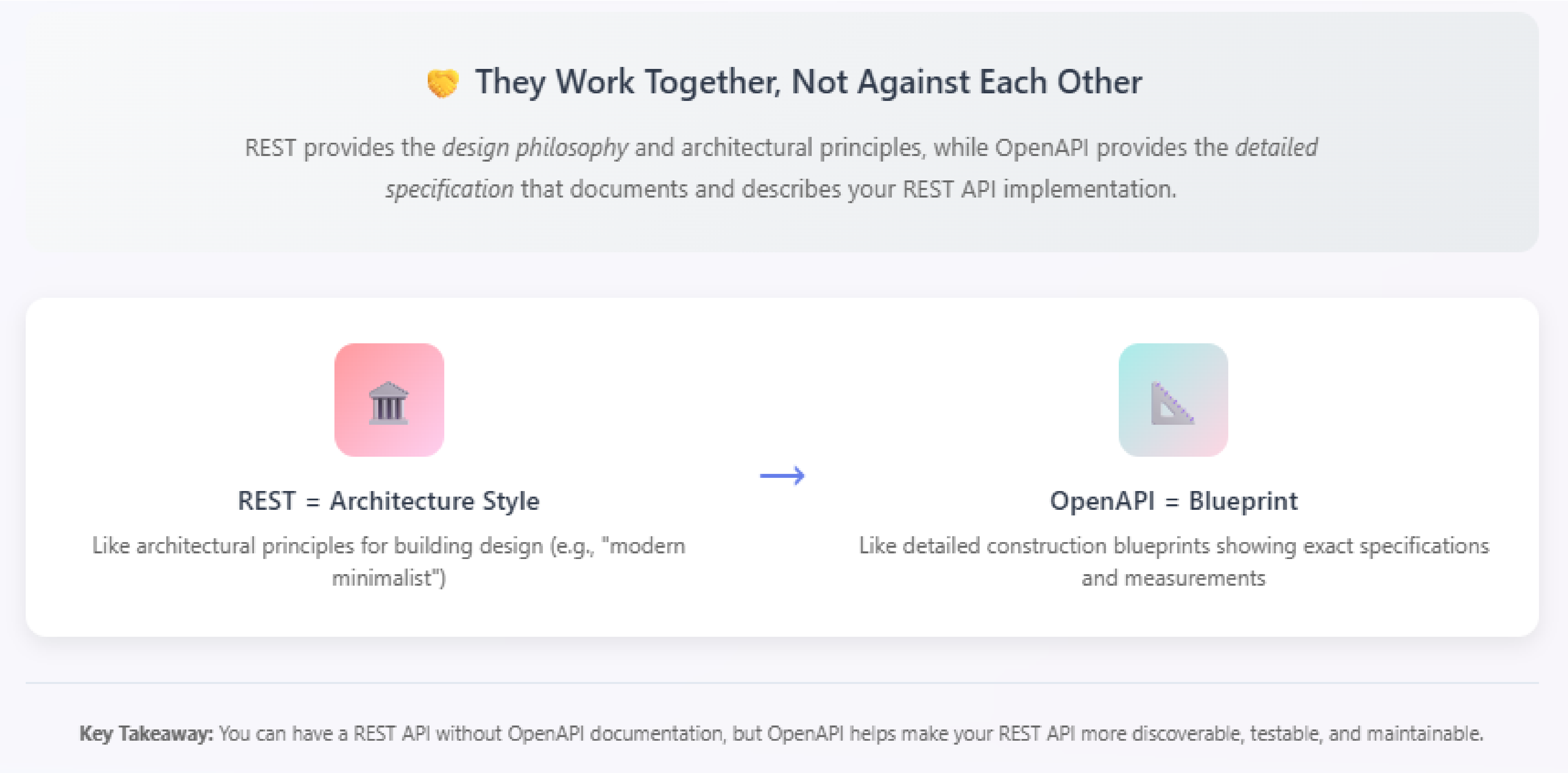
An **architectural style** that defines principles for designing web services that are scalable, stateless, and uniform.

- ✓ Architectural principles & constraints
- ✓ Stateless communication
- ✓ Resource-based URLs
- ✓ HTTP methods (GET, POST, PUT, DELETE)
- ✓ Cacheable responses

OpenAPI Specification

A **documentation standard** that provides a machine-readable contract describing how your API works in detail.

- ✓ API documentation format
- ✓ Endpoint definitions
- ✓ Request/response schemas
- ✓ Authentication methods
- ✓ Code generation capabilities



🤝 They Work Together, Not Against Each Other

REST provides the *design philosophy* and architectural principles, while OpenAPI provides the *detailed specification* that documents and describes your REST API implementation.

🏛

REST = Architecture Style

Like architectural principles for building design (e.g., "modern minimalist")

→

📐

OpenAPI = Blueprint

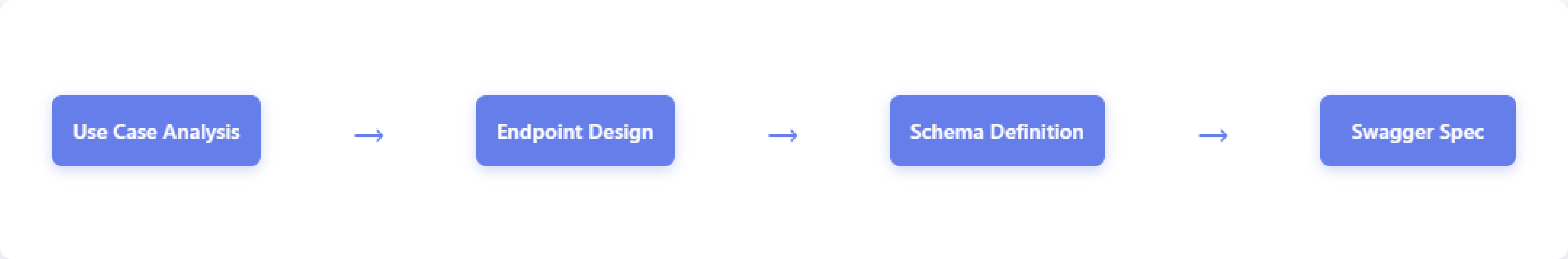
Like detailed construction blueprints showing exact specifications and measurements

Key Takeaway: You can have a REST API without OpenAPI documentation, but OpenAPI helps make your REST API more discoverable, testable, and maintainable.

API Specs Planning with Swagger

From Use Cases to Production-Ready API Documentation

Using OpenAPI Specification & Swagger Toolchain



Use Case Analysis → Endpoint Design → Schema Definition → Swagger Spec

Training Objective

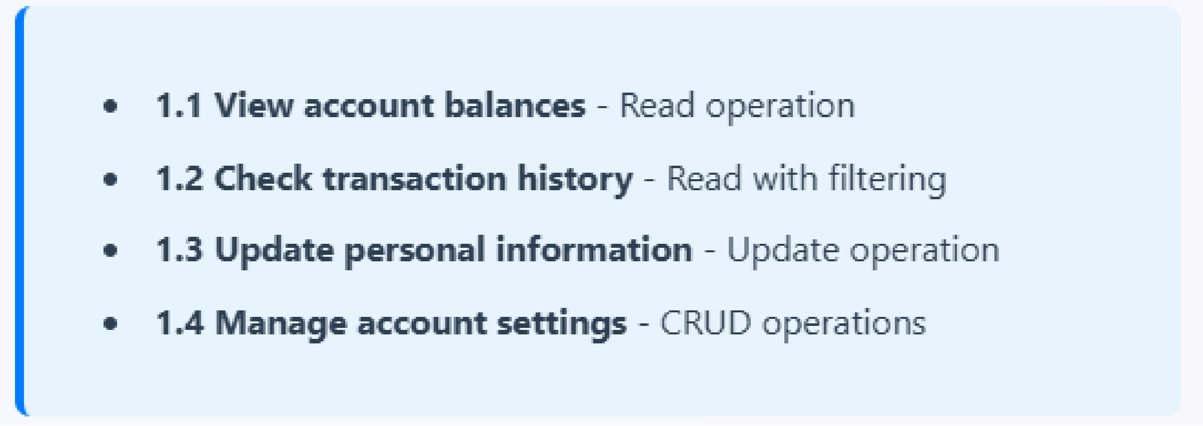
Learn to convert business use cases into well-structured API specifications systematically

OpenAPI Specification (OAS) format and **Swagger tools** for documentation, testing, and code generation, following industry best practices for REST API design.

Note: OpenAPI = Specification format | Swagger = Toolchain (Swagger UI, Swagger Editor, Swagger Codegen)

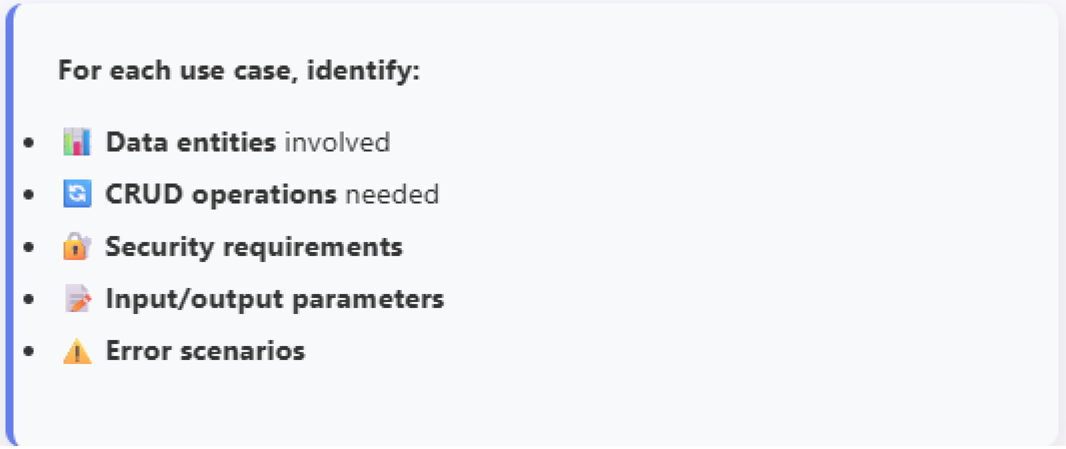
Step 1: Use Case Analysis

Given Use Case: Account Management



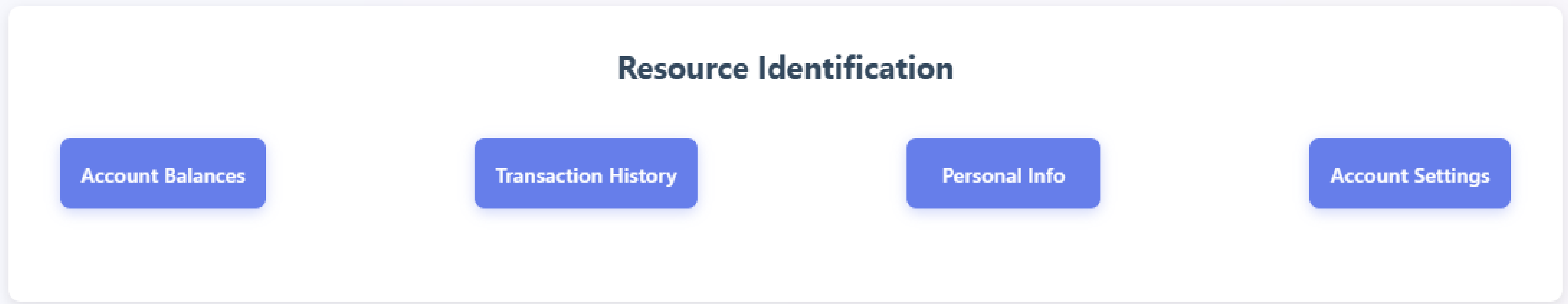
- **1.1 View account balances** - Read operation
- **1.2 Check transaction history** - Read with filtering
- **1.3 Update personal information** - Update operation
- **1.4 Manage account settings** - CRUD operations

Analysis Framework



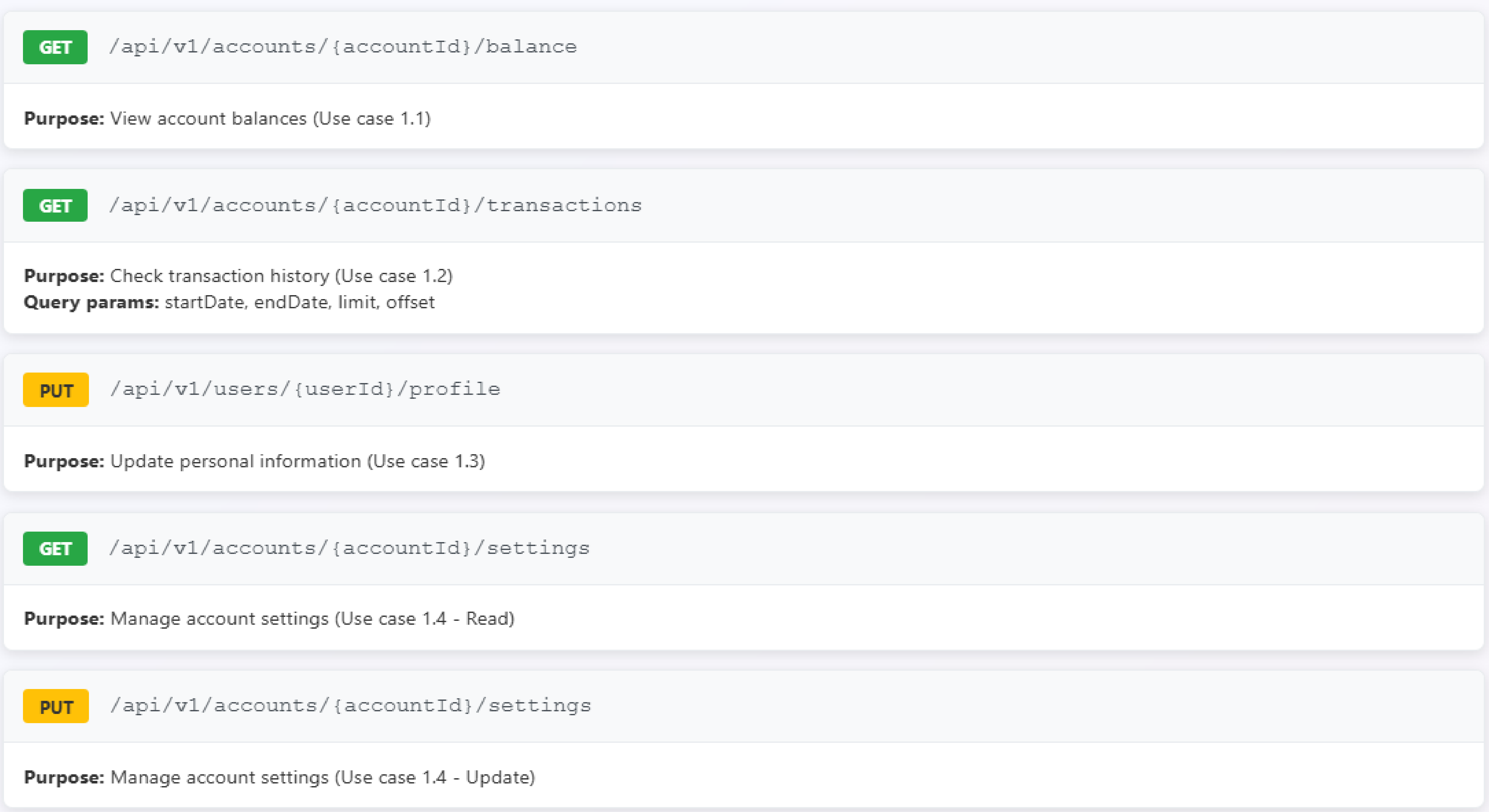
For each use case, identify:

- 📊 **Data entities** involved
- 🔄 **CRUD operations** needed
- 🔒 **Security requirements**
- 📝 **Input/output parameters**
- ⚠️ **Error scenarios**



Resource Identification

Step 2: REST Endpoint Design



GET `/api/v1/accounts/{accountId}/balance`

Purpose: View account balances (Use case 1.1)

GET `/api/v1/accounts/{accountId}/transactions`

Purpose: Check transaction history (Use case 1.2)
Query params: startDate, endDate, limit, offset

PUT `/api/v1/users/{userId}/profile`

Purpose: Update personal information (Use case 1.3)

GET `/api/v1/accounts/{accountId}/settings`

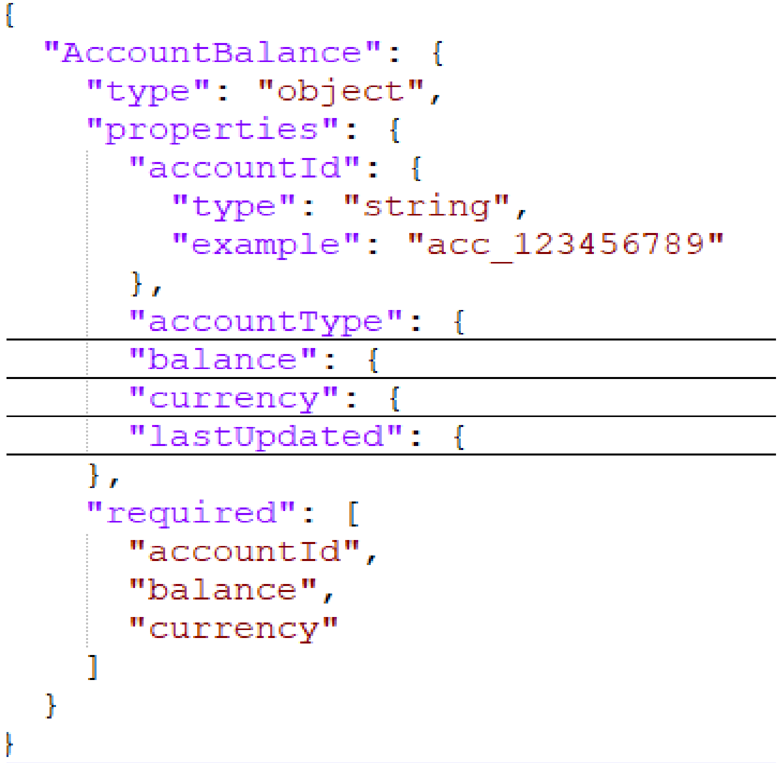
Purpose: Manage account settings (Use case 1.4 - Read)

PUT `/api/v1/accounts/{accountId}/settings`

Purpose: Manage account settings (Use case 1.4 - Update)

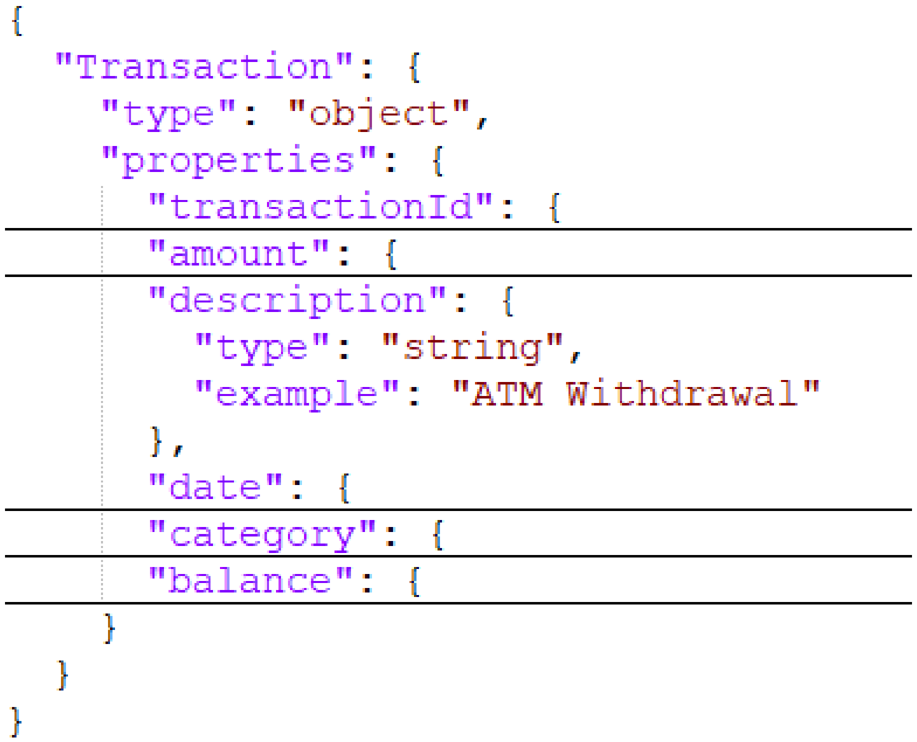
Step 3: Data Models & Schemas

Account Balance Model

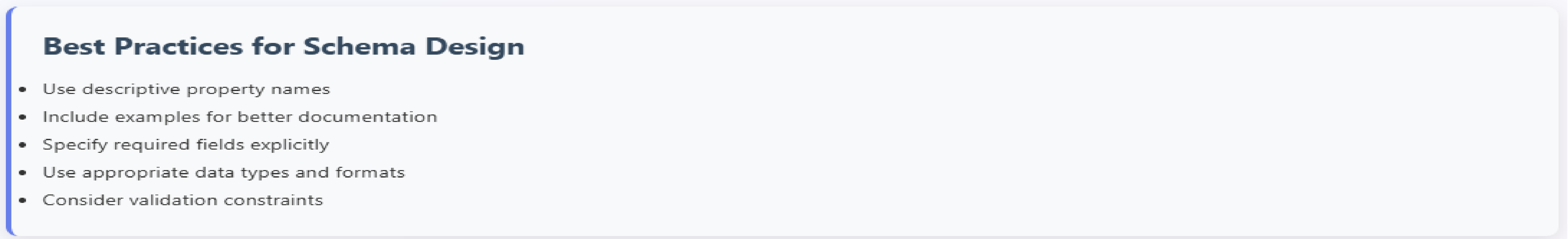


```
{
  "AccountBalance": {
    "type": "object",
    "properties": {
      "accountId": {
        "type": "string",
        "example": "acc_123456789"
      },
      "accountType": {
      "balance": {
      "currency": {
      "lastUpdated": {
    },
    "required": [
      "accountId",
      "balance",
      "currency"
    ]
  }
}
```

Transaction Model



```
{
  "Transaction": {
    "type": "object",
    "properties": {
      "transactionId": {
      "amount": {
      "description": {
        "type": "string",
        "example": "ATM Withdrawal"
      },
      "date": {
      "category": {
      "balance": {
    }
  }
}
```

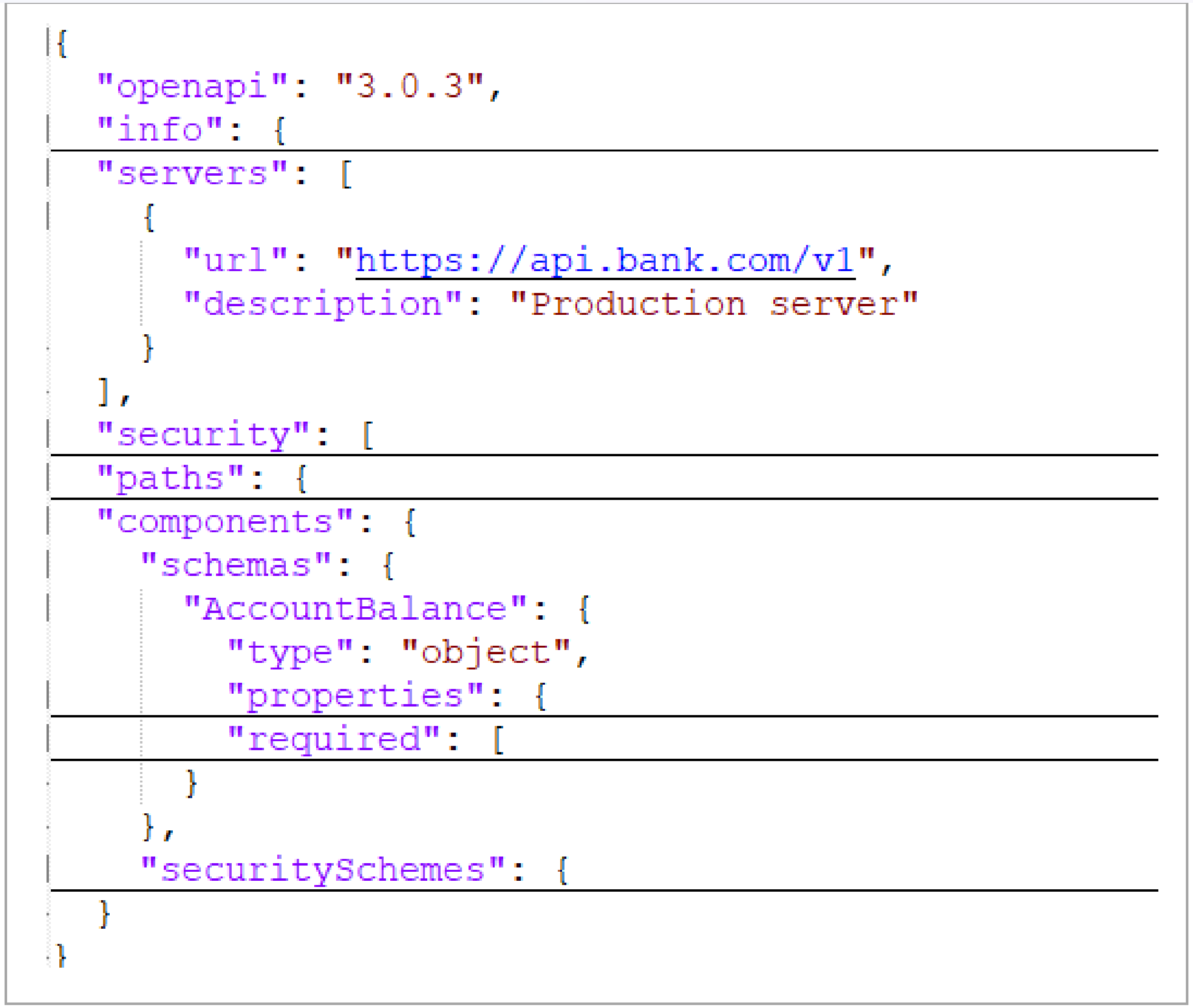


Best Practices for Schema Design

- Use descriptive property names
- Include examples for better documentation
- Specify required fields explicitly
- Use appropriate data types and formats
- Consider validation constraints

Step 4: Complete OpenAPI Specification (for Swagger)

This YAML/JSON spec will be consumed by Swagger tools

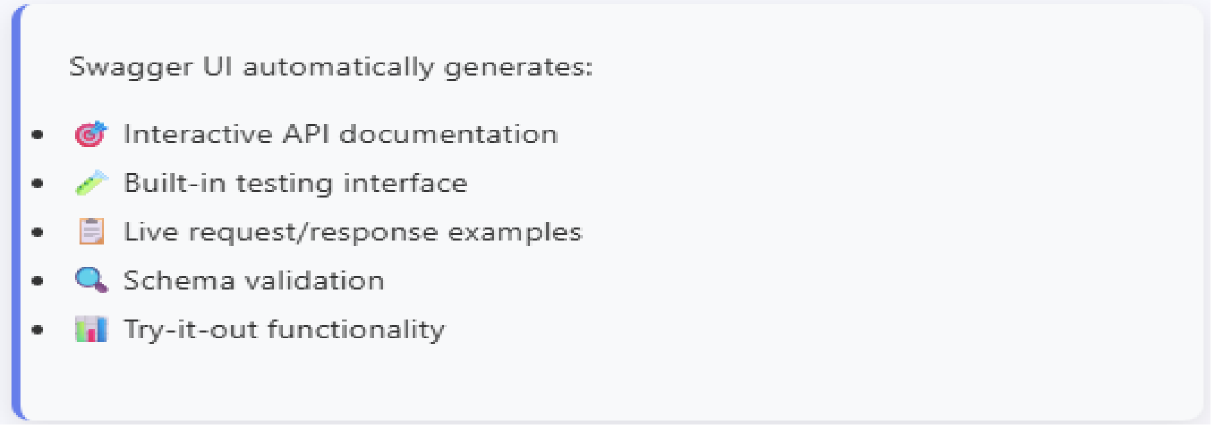


```
{
  "openapi": "3.0.3",
  "info": {
  "servers": [
    {
      "url": "https://api.bank.com/v1",
      "description": "Production server"
    }
  ],
  "security": [
  "paths": {
  "components": {
    "schemas": {
      "AccountBalance": {
        "type": "object",
        "properties": {
        "required": [
      }
    },
    "securitySchemes": {
  }
}
```

Complete_OpenAI_Spec.json

Step 5: Swagger Tools for Testing & Validation

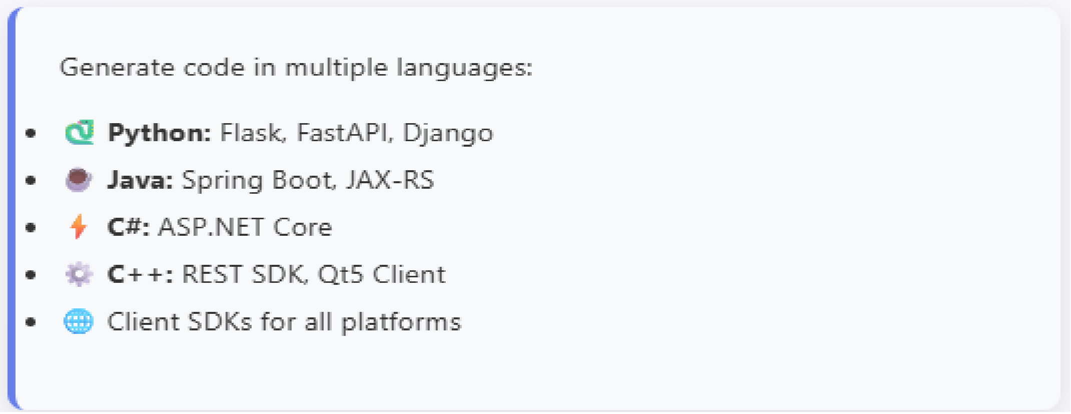
Swagger UI - Interactive Documentation



Swagger UI automatically generates:

- 🎯 Interactive API documentation
- 🧪 Built-in testing interface
- 📋 Live request/response examples
- 🔍 Schema validation
- 📊 Try-it-out functionality

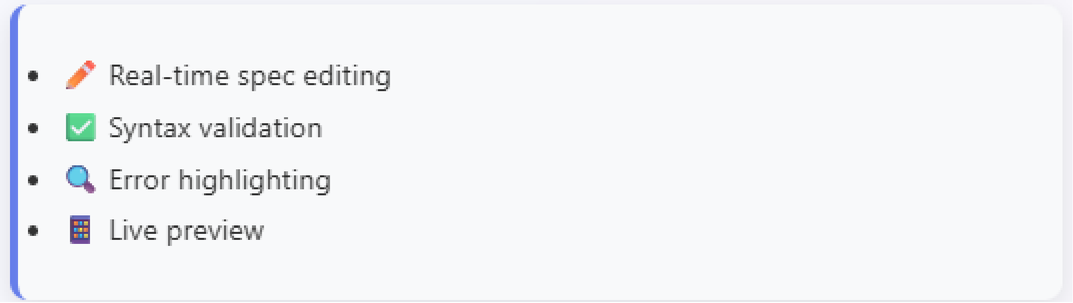
Swagger Codegen



Generate code in multiple languages:

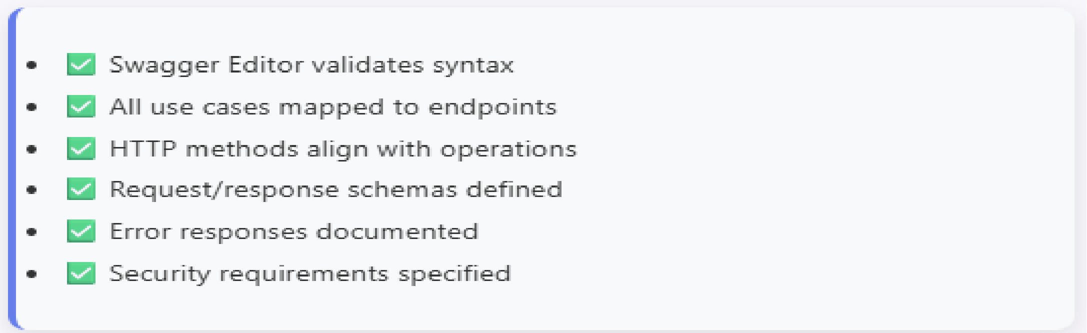
- 🐍 **Python:** Flask, FastAPI, Django
- ☕ **Java:** Spring Boot, JAX-RS
- ⚡ **C#:** ASP.NET Core
- ⚙️ **C++:** REST SDK, Qt5 Client
- 🌐 Client SDKs for all platforms

Swagger Editor



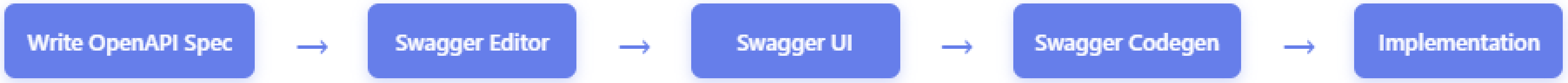
- ✏️ Real-time spec editing
- ✅ Syntax validation
- 🔍 Error highlighting
- 📱 Live preview

Validation Checklist

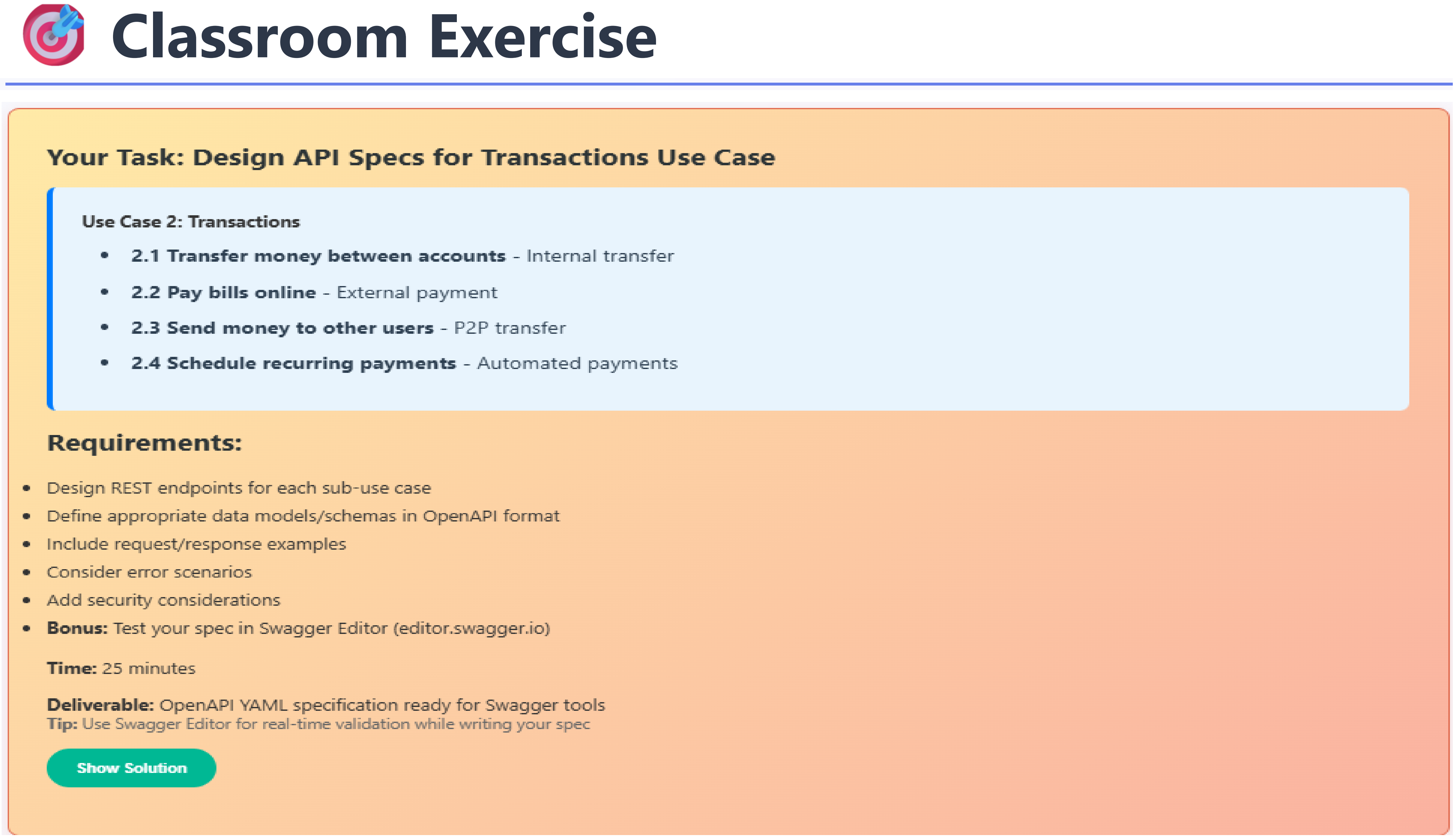


- ✅ Swagger Editor validates syntax
- ✅ All use cases mapped to endpoints
- ✅ HTTP methods align with operations
- ✅ Request/response schemas defined
- ✅ Error responses documented
- ✅ Security requirements specified

Swagger-Powered Development Workflow



Tools URLs: editor.swagger.io | swagger.io/tools/swagger-ui | swagger.io/tools/swagger-codegen



Classroom Exercise

Your Task: Design API Specs for Transactions Use Case

Use Case 2: Transactions

- **2.1 Transfer money between accounts** - Internal transfer
- **2.2 Pay bills online** - External payment
- **2.3 Send money to other users** - P2P transfer
- **2.4 Schedule recurring payments** - Automated payments

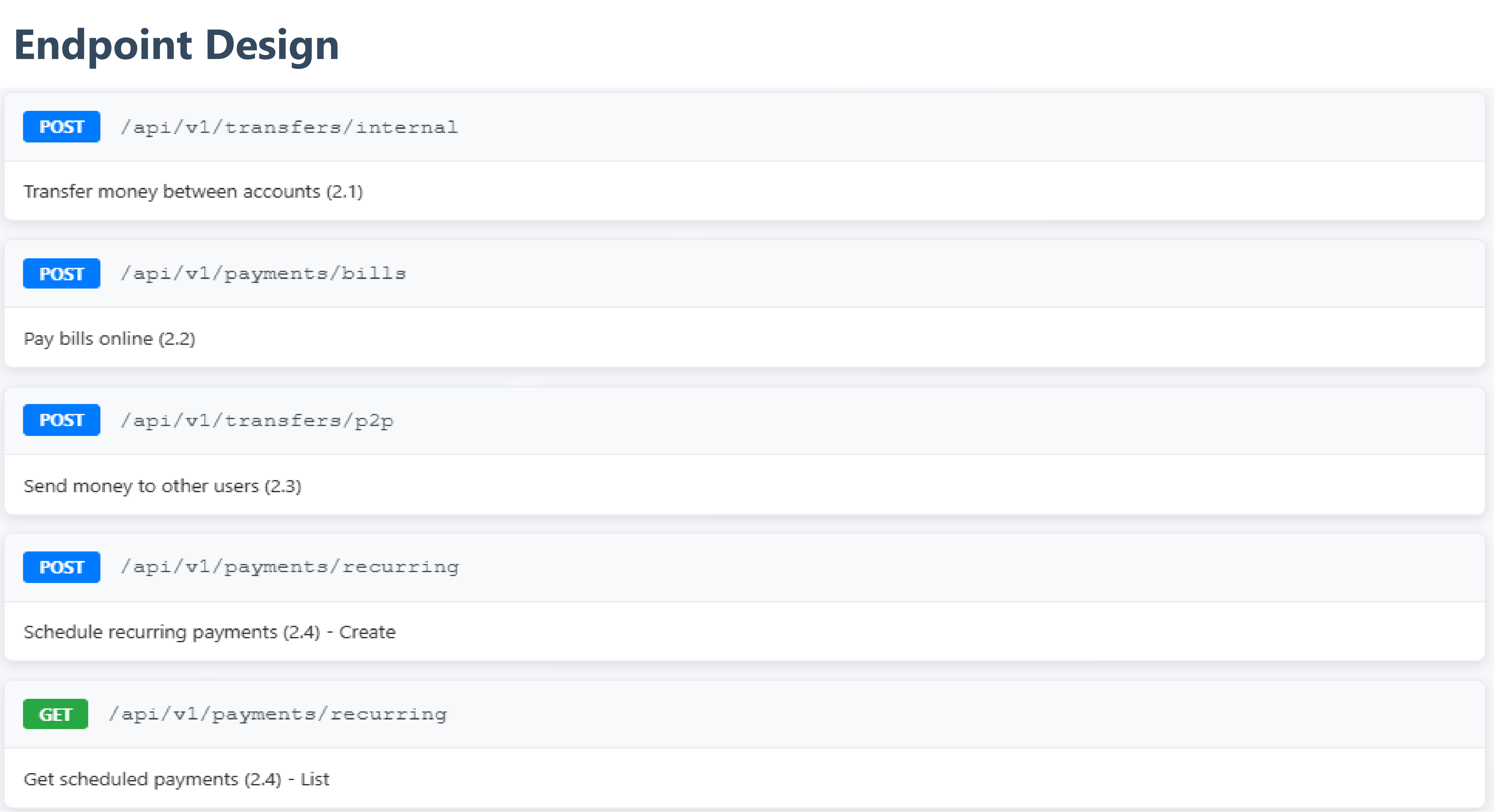
Requirements:

- Design REST endpoints for each sub-use case
- Define appropriate data models/schemas in OpenAPI format
- Include request/response examples
- Consider error scenarios
- Add security considerations
- **Bonus:** Test your spec in Swagger Editor (editor.swagger.io)

Time: 25 minutes

Deliverable: OpenAPI YAML specification ready for Swagger tools
Tip: Use Swagger Editor for real-time validation while writing your spec

💡 Exercise Solution



Endpoint Design

POST `/api/v1/transfers/internal`

Transfer money between accounts (2.1)

POST `/api/v1/payments/bills`

Pay bills online (2.2)

POST `/api/v1/transfers/p2p`

Send money to other users (2.3)

POST `/api/v1/payments/recurring`

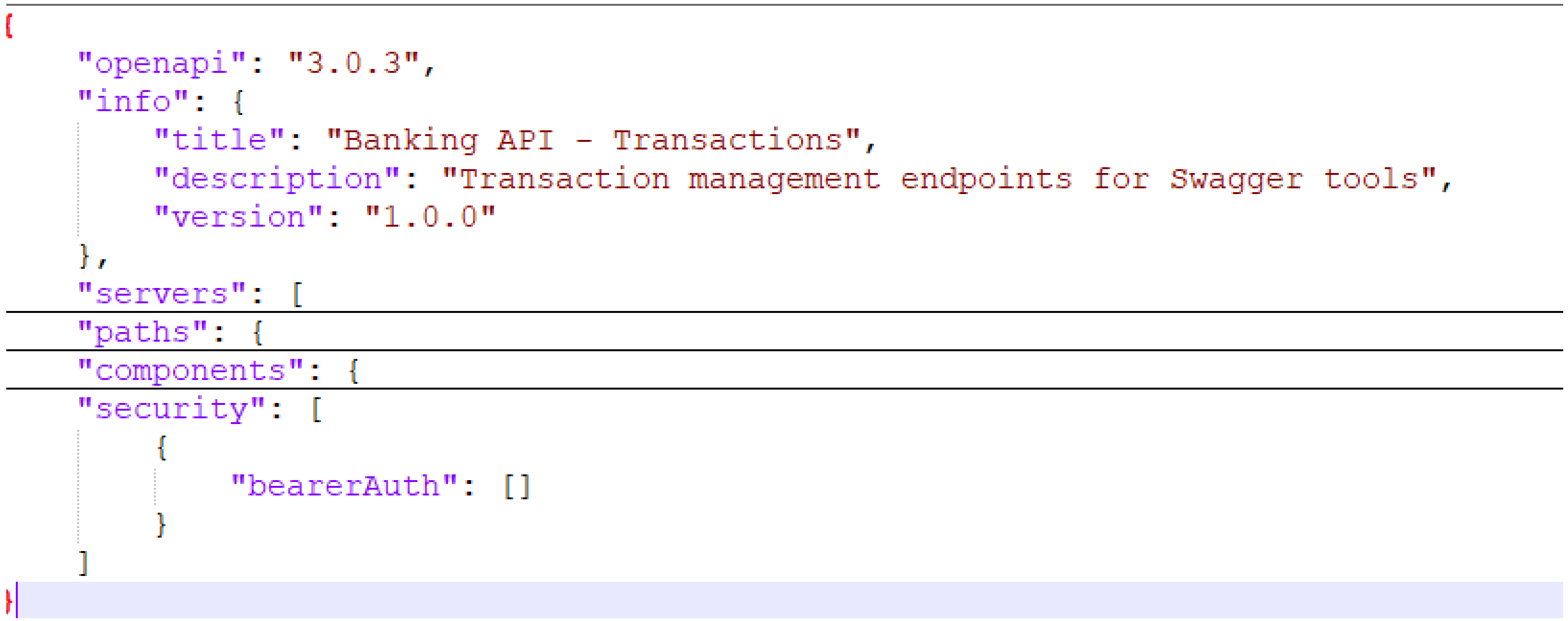
Schedule recurring payments (2.4) - Create

GET `/api/v1/payments/recurring`

Get scheduled payments (2.4) - List

💡 Exercise Solution

Sample Complete Spec for Swagger Tools



```
{
    "openapi": "3.0.3",
    "info": {
        "title": "Banking API - Transactions",
        "description": "Transaction management endpoints for Swagger tools",
        "version": "1.0.0"
    },
    "servers": [
    "paths": {
    "components": {
    "security": [
        {
            "bearerAuth": []
        }
    ]
}
```

💡 **Pro Tip:** Copy this spec to **editor.swagger.io** to see live documentation, test endpoints, and generate client code in C++, Python, Java, or C#!

Appendix