# Welcome to this CoGrammar Lecture:

# Use Case Analysis and Sequence Diagrams

## The session will start shortly...

Questions? Drop them in the chat.
We'll have dedicated moderators
answering questions.

# Software Engineering Session Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly. **(Fundamental British Values: Mutual Respect and Tolerance)**
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** throughout this session, should you wish to ask any follow-up questions.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Academic Sessions. You can submit these questions here: **Questions**

# **Software Engineering Session Housekeeping** cont.

- For all **non-academic questions**, please submit a query: **www.hyperiondev.com/support**
- Report a **safeguarding** incident: **www.hyperiondev.com/safeguardreporting**
- We would love your **feedback** on lectures: **Feedback on Lectures**
- If you are hearing impaired, please kindly use your computer's function through Google chrome to enable captions.

# Safeguarding & Welfare

We are committed to all our students and staff feeling safe and happy; we want to make sure there is always someone you can turn to if you are worried about anything.

If you are feeling upset or unsafe, are worried about a friend, student or family member, or you feel like something isn't right, speak to our safeguarding team:

Ian Wyles
Designated Safeguarding Lead

Simone Botes

Nurhaan Snyman

Rafiq Manan

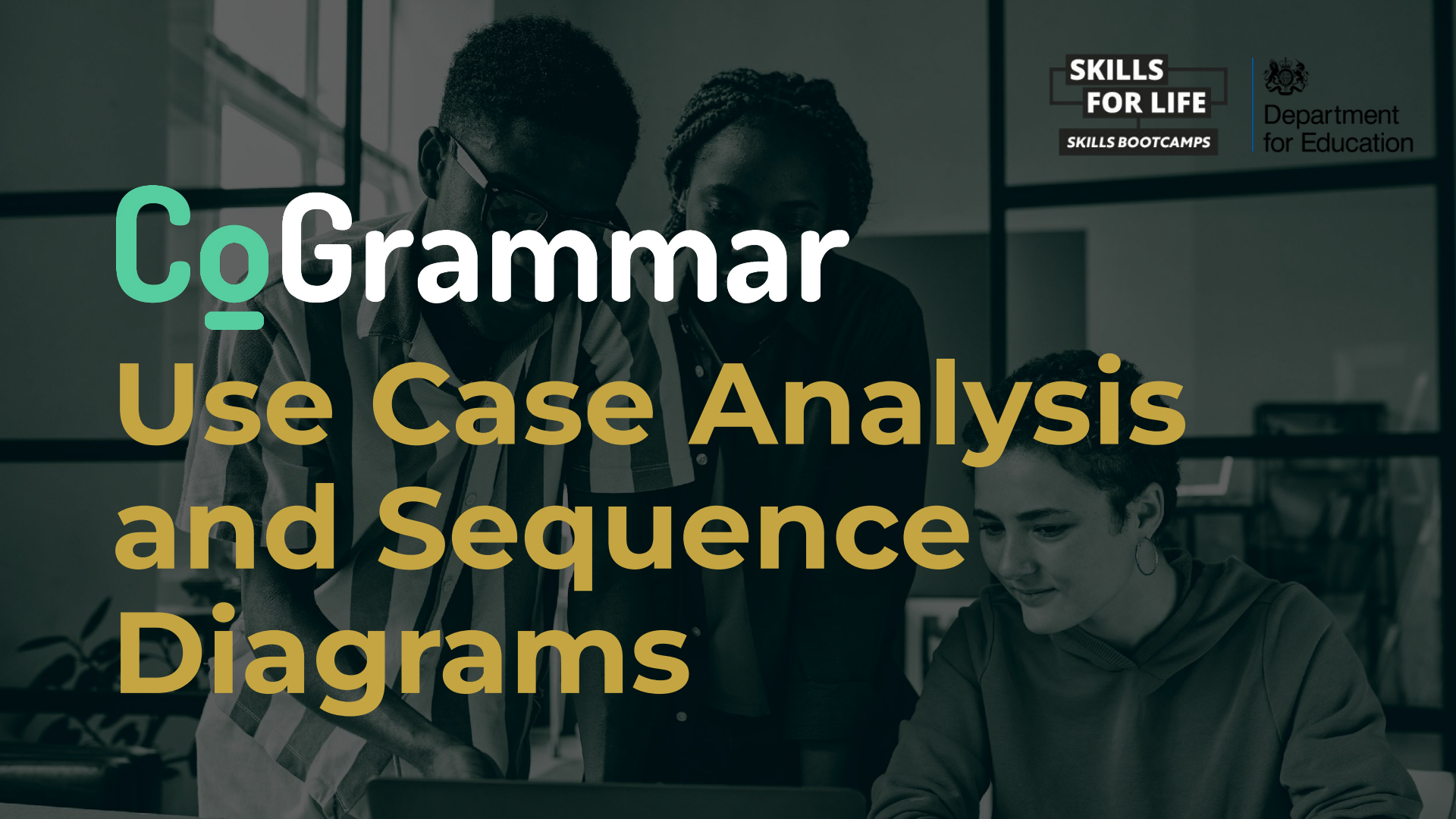
Ronald Munodawafa

Tevin Pitts

**Scan to report a safeguarding concern**

or email the Designated Safeguarding Lead:
Ian Wyles
safeguarding@hyperiondev.com

CoGrammar HyperionDev

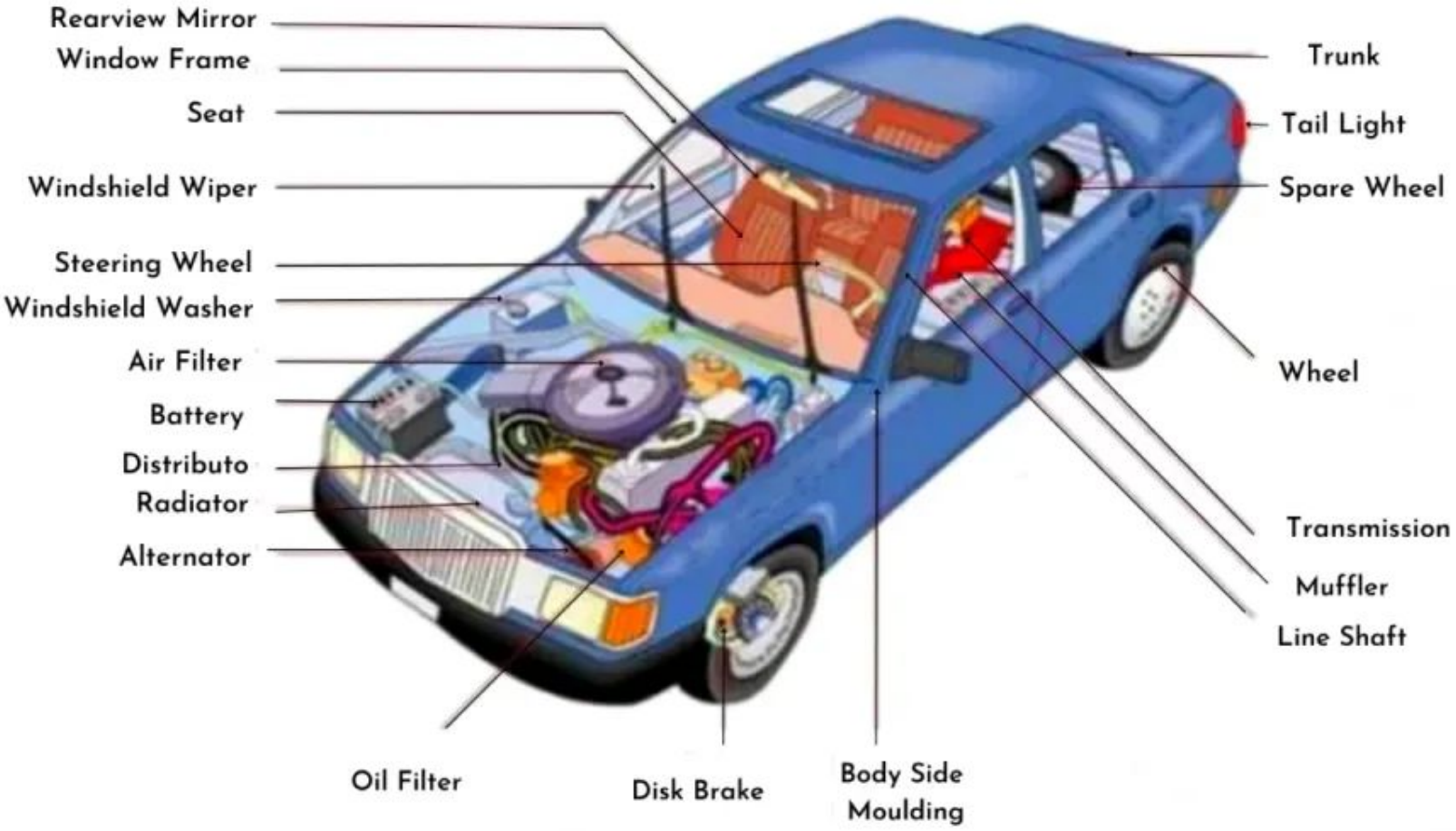

SKILLS FOR LIFE
SKILLS BOOTCAMPS
Department for Education
CoGrammar
Use Case Analysis and Sequence Diagrams

# Learning Objectives & Outcomes

- Explain the significance of modularisation, including benefits like maintainability and scalability, and apply techniques such as object-oriented design in project development,
- Describe sequence diagrams' role in visualising system interactions, analyse them for system dynamics, and create diagrams for various use cases,
- Demonstrate use case analysis for capturing user requirements, including identifying actors and scenarios, and prioritise use cases using elicitation techniques.

SKILLS FOR LIFE
SKILLS BOOTCAMPS
Department for Education
CoGrammar
Modular Programming

# PARTS OF A CAR



facebook

Email or phone | Password | Log in | Forgotten account?

# Marketplace

Search Marketplace

- Browse all
- Your account

+ Create new listing

Filters

San Francisco, California

Categories

- Vehicles
- Property for rent
- Classifieds
- Clothing
- Electronics
- Entertainment

## Today's picks

San Francisco · 65 km

$350
Wooden Gazebo 10X10
Antioch, CA

$25,000
1963 Ford F100
Livermore, CA

FREE
Weight
San Francisco, CA

$18,000
1999 Ford 450
San Pablo, CA

**Log in or sign up for Facebook to connect with friends, family and people you know.**

Log in | or | Create New Account

# Poll

Given the image of Facebook Marketplace and a vehicle, what are the similarities between them?

a. Maintainable components
b. Reusable components
c. Monolithic design
d. Adaptability to new working components
e. Coupled components
f. Testable components

# Intuition

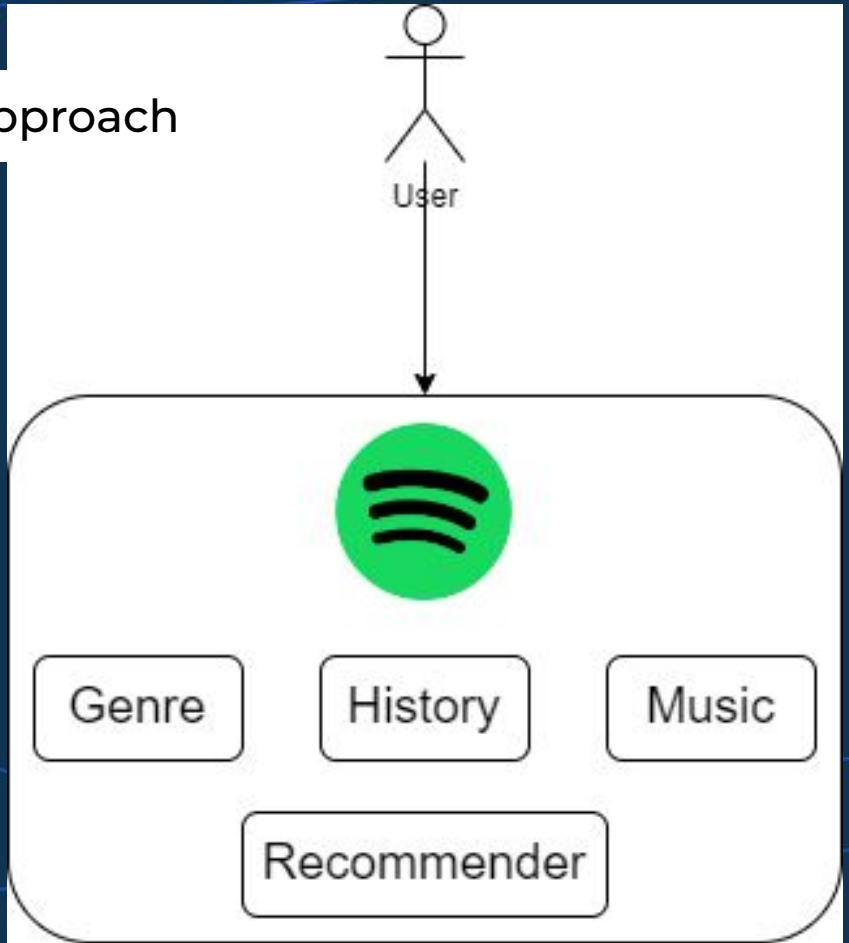
In Facebook Marketplace's intricate **codebase**, modularity functions like a network of buildings in a cityscape. If a glitch arises in a **specific feature**, such as photo posting, the modular design allows engineers to target and rectify the issue within **the relevant modules without bringing down the entire platform**. This approach ensures that Facebook remains resilient and adaptable, minimizing disruptions to user experience while maintaining the integrity of the system.

# Definition and Importance

- Modularity in software design is a structured approach that aims to streamline complexity **by breaking down systems into distinct, reusable modules**.
- These **modules encapsulate specific functionality, allowing for easier extension, modification, and integration** into various contexts.

# Use Case: Spotify

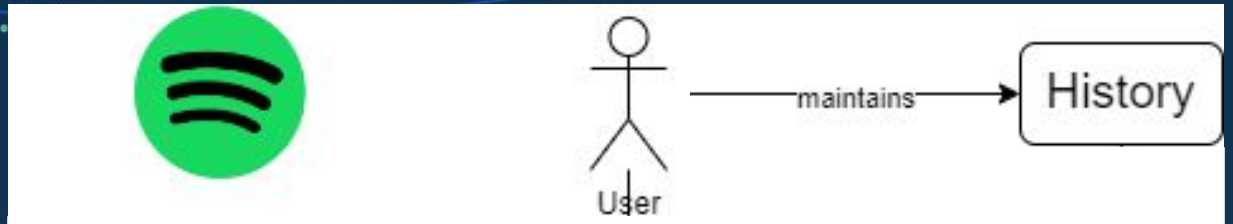
Monolithic approach




CoGrammar

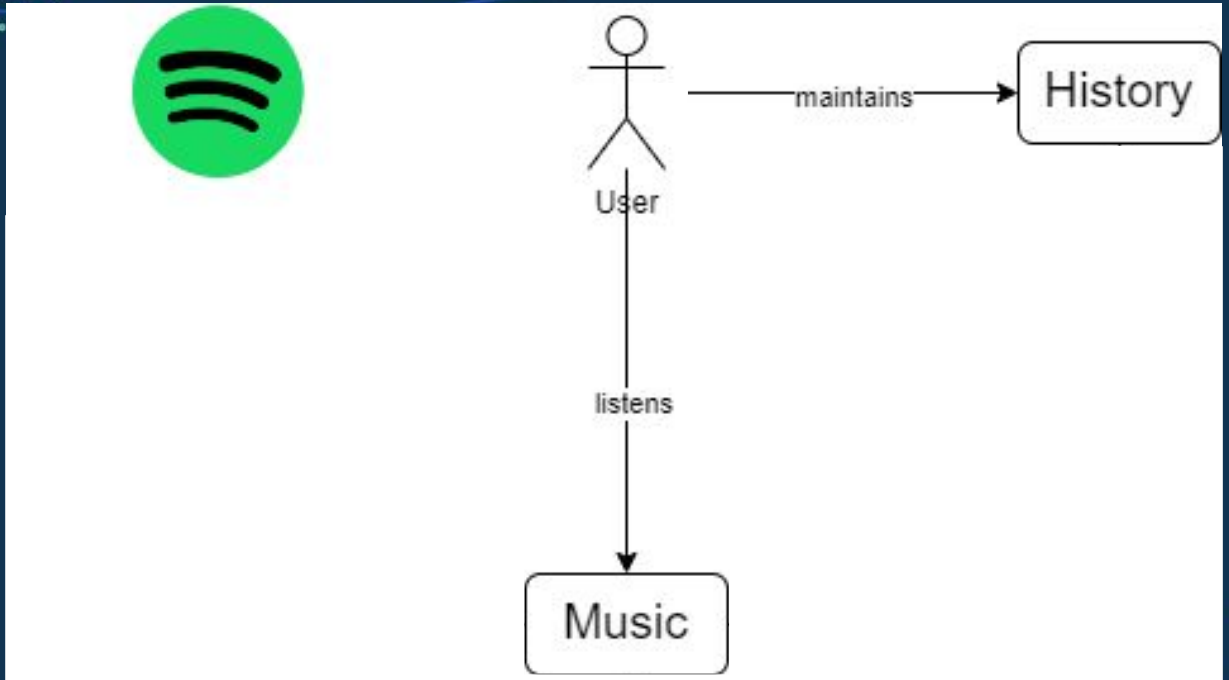
# Use Case: Spotify

# Use Case: Spotify



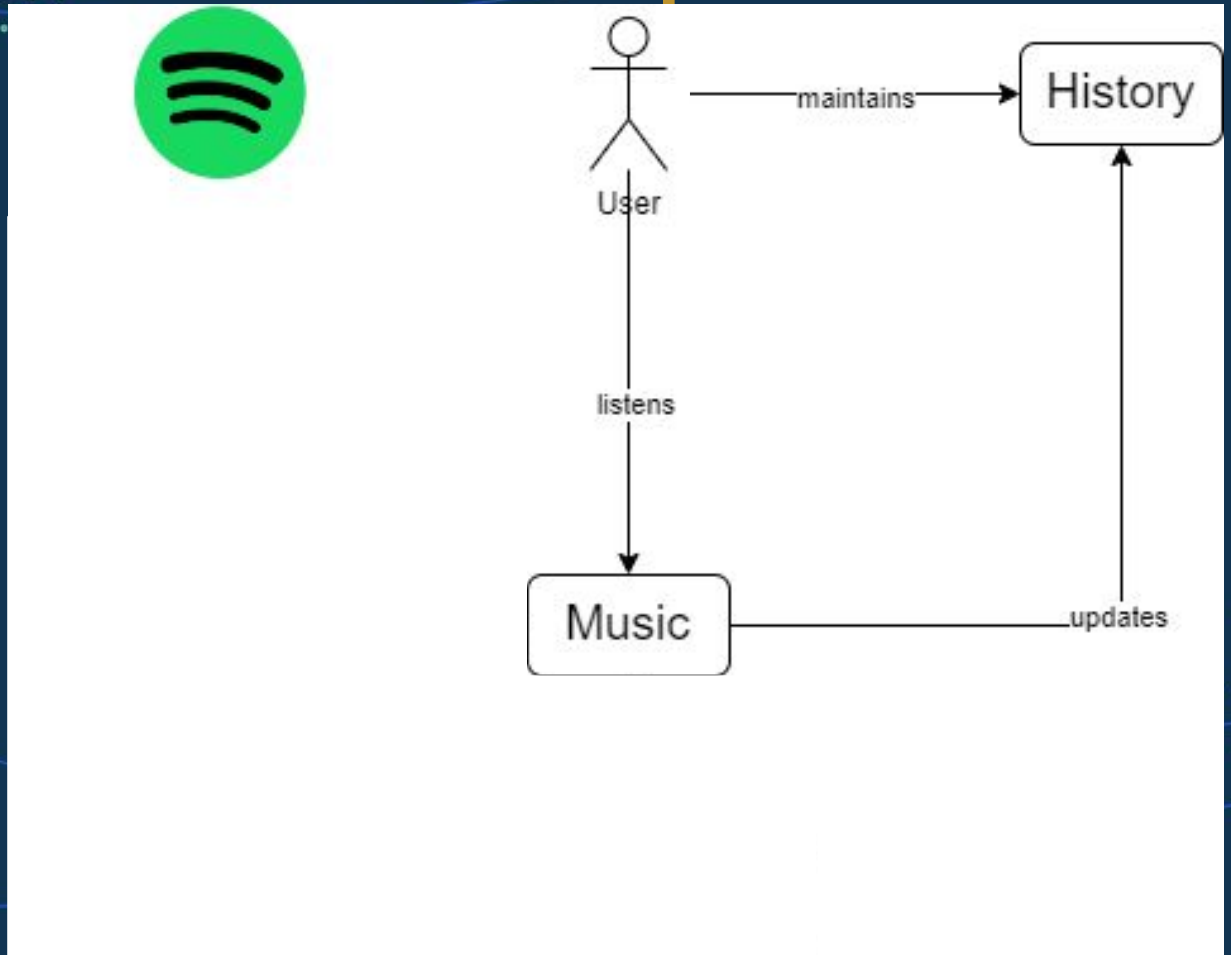

CoGrammar

# Use Case: Spotify



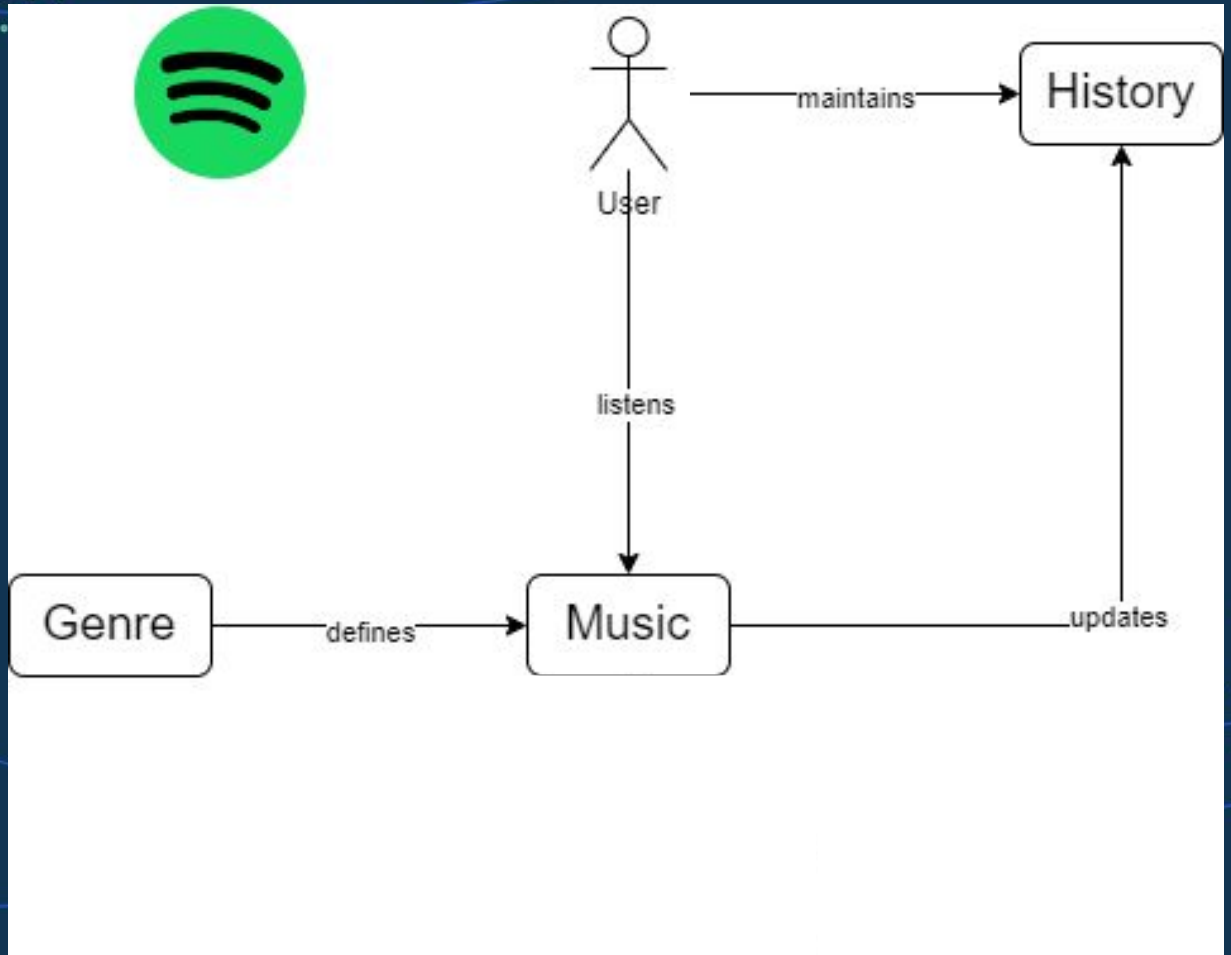

CoGrammar

# Definition and Importance



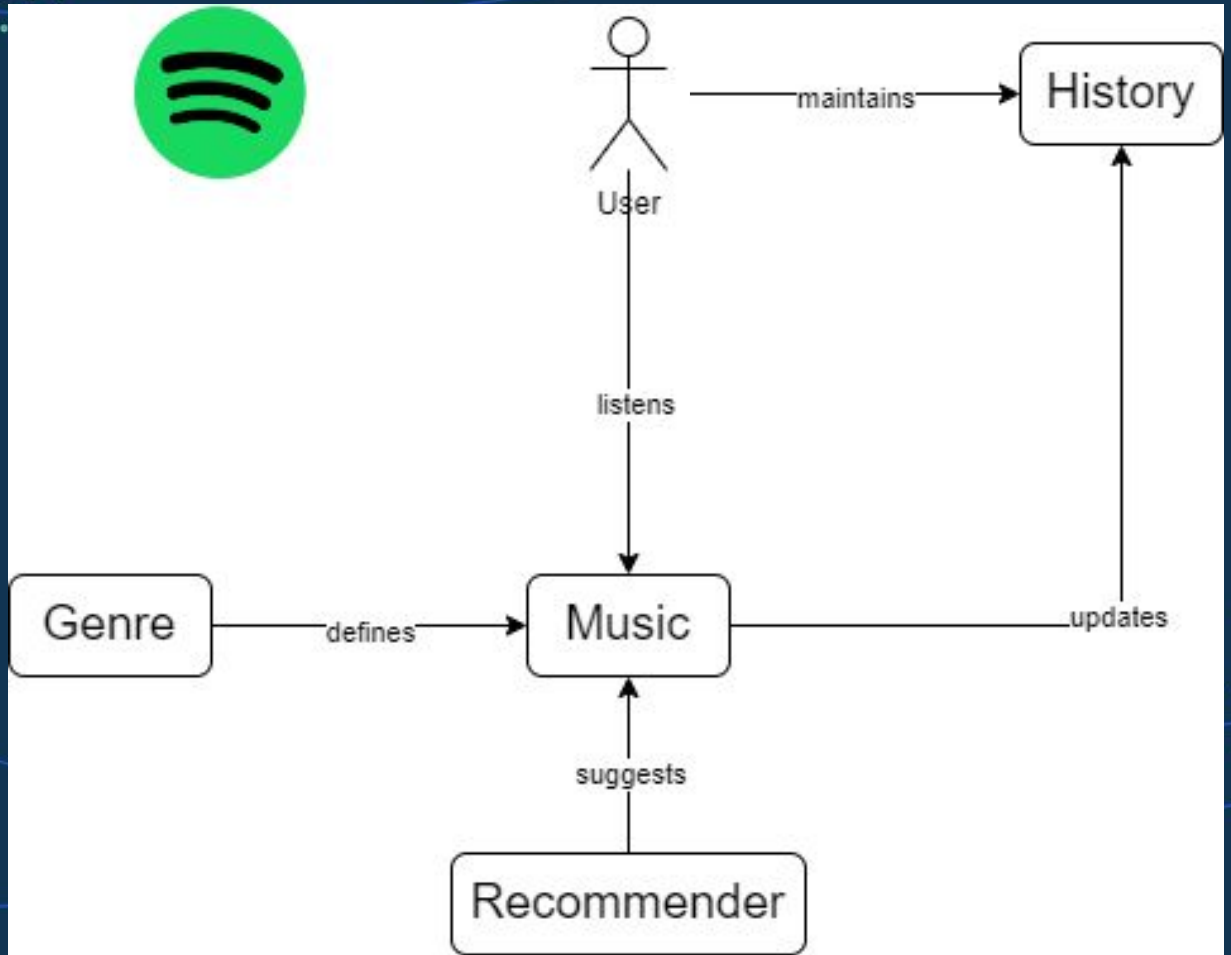

CoGrammar

# Use Case: Spotify




CoGrammar

# Use Case: Spotify

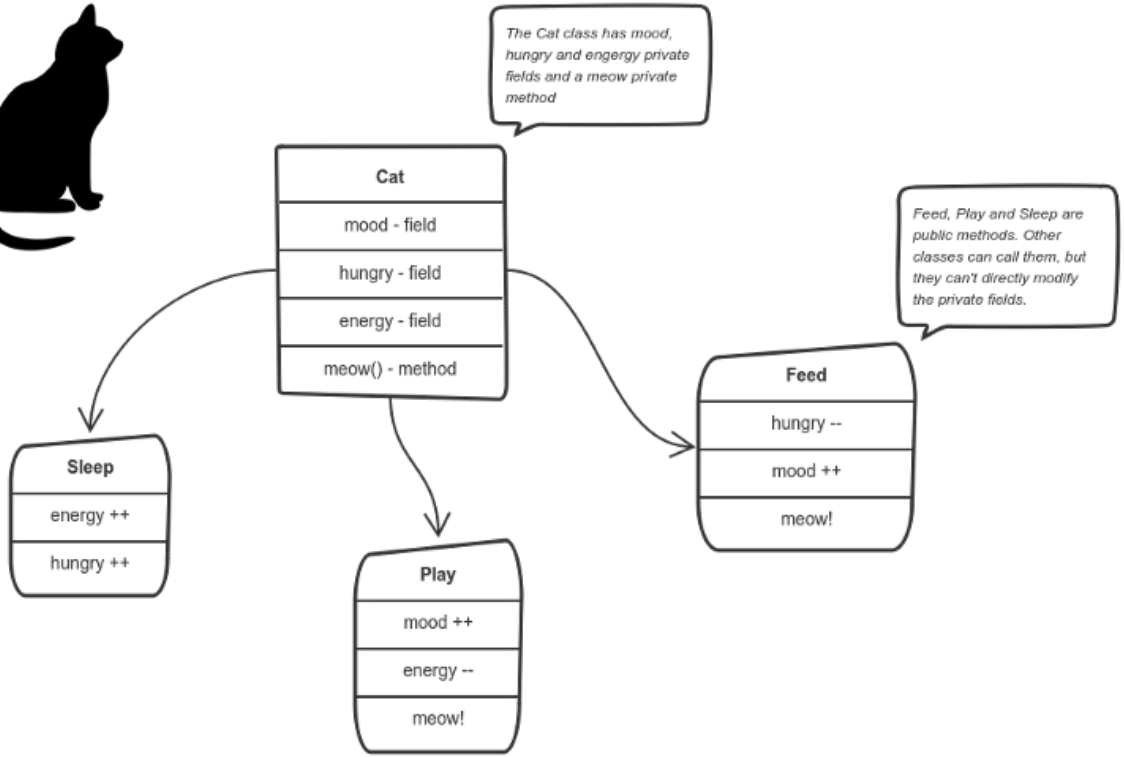



CoGrammar

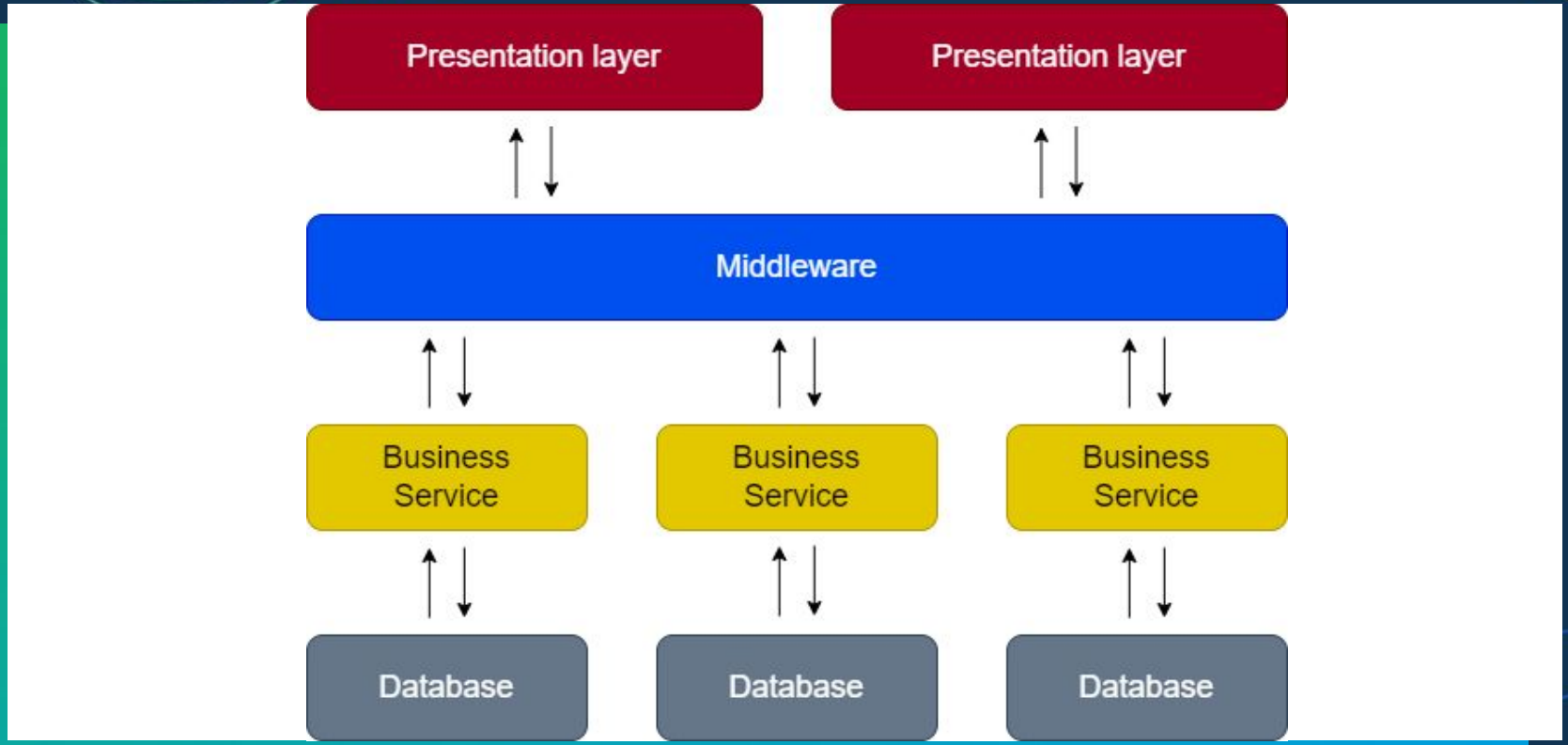
# Techniques and Approaches

- This design principle also enhances portability, as modules can be transferred across different environments or platforms with minimal adjustments.
- Modularity promotes maintainability by isolating changes and facilitating debugging and updates, ultimately improving the robustness and flexibility of software systems.

# Object Oriented Programming



# Service Oriented Architecture (SOA)



# Cohesion and Coupling

- Coupling describes the relationships between modules, and cohesion describes the relationships within them.
- Coupling is the measure of the degree of interdependence between the modules. In a good design, the various component parts (e.g. the classes) have low coupling.
- Cohesion is the measure of strength of relationships between pieces of functionality within a module. In a good design, the various modules have high cohesion.

# Cohesion

```
# Low Cohesion Code Example
class Car:
    def __init__(self, make, model,
year):
        self.make = make
        self.model = model
        self.year = year

    def display_details(self):
        print(f"""Make: {self.make},
              Model: {self.model},
              Year: {self.year}""")

    def calculate_price(self):
        # Calculate car price
        pass

    def update_inventory(self):
        # Update car inventory in database
        pass

    def send_notification(self):
        # Send notification to user
        pass
```

```
# High Cohesion Code Example
class Car:
    def __init__(self, make, model,
year):
        self.make = make
        self.model = model
        self.year = year

    def display_make(self):
        print(f"Make: {self.make}")

    def display_model(self):
        print(f"Model: {self.model}")

    def display_year(self):
        print(f"Year: {self.year}")

    def display_details(self):
        print(f"""Make: {self.make},
              Model: {self.model},
              Year: {self.year}
              """)
```

# Coupling

```python
# Low Coupling Code Example
class Bank:
    def __init__(self, customer_database, account_manager):
        self.customer_database = customer_database
        self.account_manager = account_manager

    def open_account(self, customer):
        customer_id = self.customer_database.add_customer(customer)
        account_number = self.account_manager.create_account(customer_id)
        return account_number
```

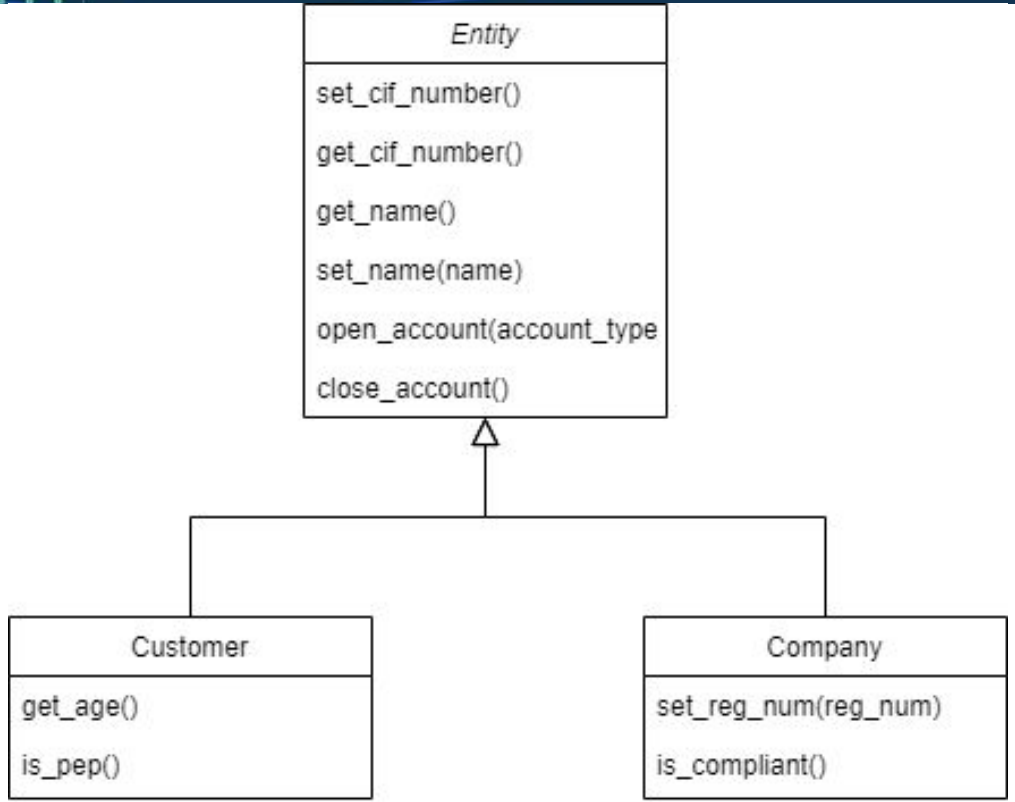
```python
# High Coupling Code Example
class Bank:
    def __init__(self):
        self.customer_database = CustomerDatabase()
        self.account_manager = AccountManager()

    def open_account(self, customer):
        customer_id = self.customer_database.add_customer(customer)
        account_number = self.account_manager.create_account(customer_id)
        return account_number
```

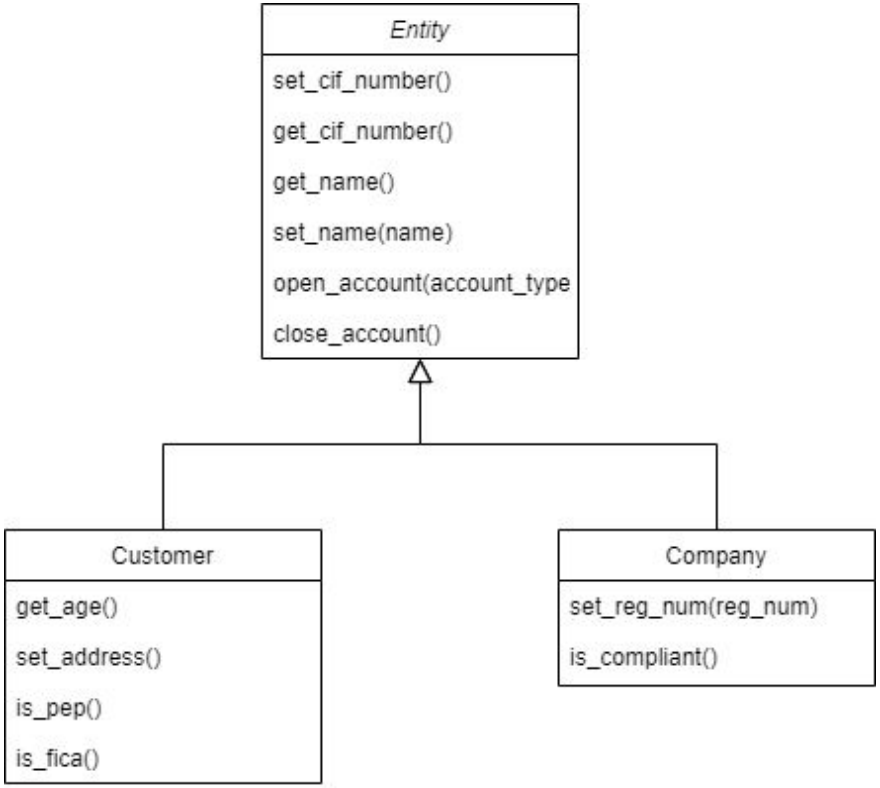
High dependency on `CustomerDatabase()` and `AccountManager()` to be created before creating a `Bank`

CoGrammar

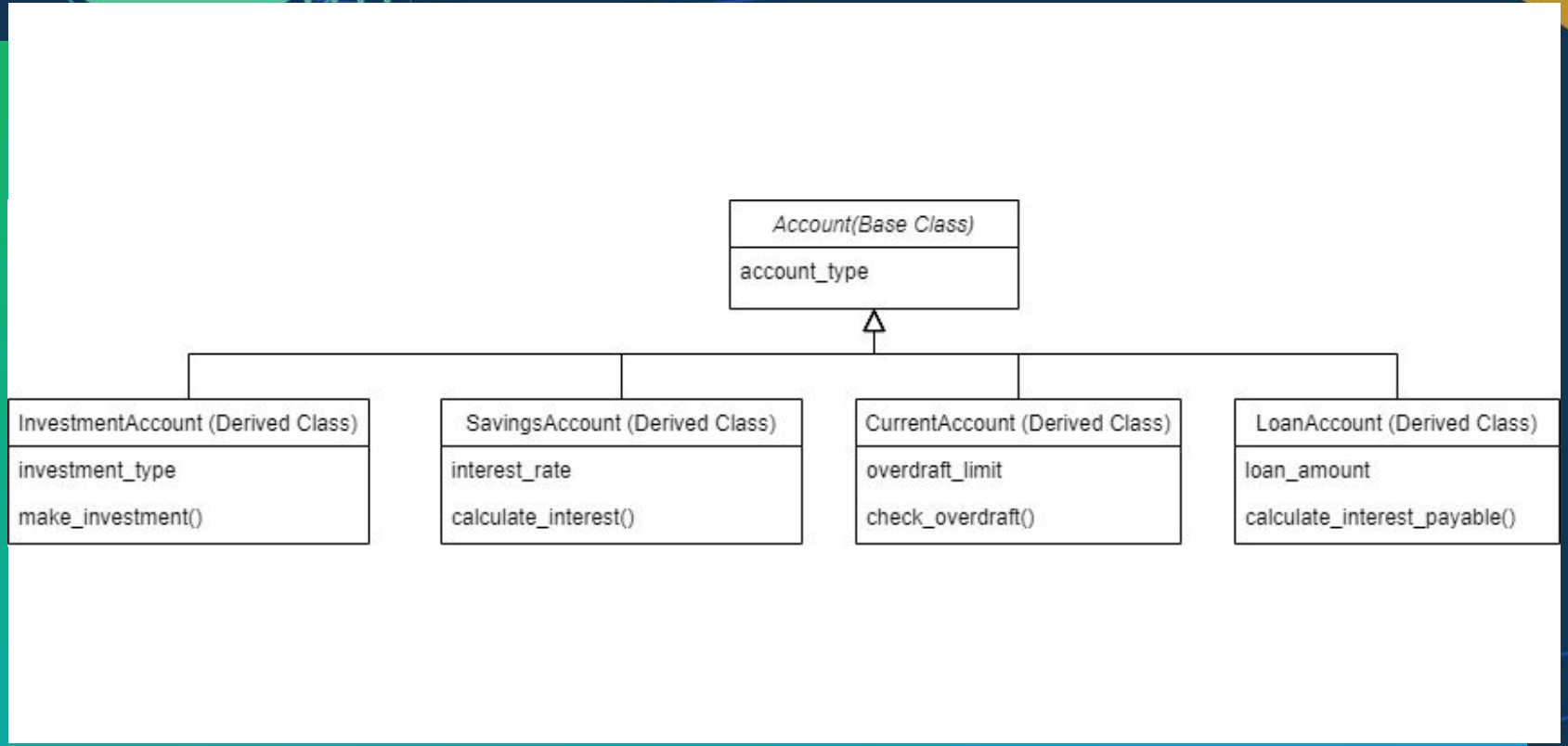
# Open/Closed Principle



# Open/Closed Principle

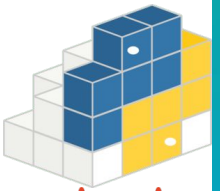


# Open/Closed Principle



# Benefits of Modularisation

- **Collaboration:** Team members can work on different modules concurrently, reducing conflicts and dependencies.
- **Maintainability:** The codebase is easier to understand, update, and debug the code. Changes or fixes can be made to individual modules without affecting other parts of the system, reducing the risk of unintended consequences.
- **Reusability:** Modular programming promotes the creation of independent modules or components that can be reused in different parts of the software system or even in other projects. This reduces duplication of code and saves development time.
- **Ease of Refactoring:** Developers can modify or improve individual modules without affecting other parts of the system, making it safer and more efficient to refactor code to improve its structure, readability, and maintainability.



# Benefits of Modularisation

- **Testing and Debugging:** Modular programming simplifies testing and debugging efforts as modules can be tested independently of each other. This modular approach allows for more focused and efficient testing, leading to higher code quality and fewer bugs in the final product.

- **Flexibility:** Modular programming enables developers to modify or replace individual modules without impacting other parts of the system. This flexibility allows for easier adaptation to changing requirements, technological advancements, or business needs over time.

- **Scalability:** Modular programming facilitates the scalability of software systems by allowing developers to add new features or functionalities as separate modules. This modular approach makes it easier to extend the system's capabilities without having to overhaul the entire codebase.

# Poll

```
def calculate(operation, x, y):
    if operation == "add":
        return x + y
    elif operation == "subtract":
        return x - y
    elif operation == "multiply":
        return x * y
    elif operation == "divide":
        if y == 0:
            return "Error: Division by
zero"
        else:
            return x / y
    else:
        return "Error: Invalid operation"

print(calculate("add", 5, 3))
print(calculate("subtract", 5, 3))
print(calculate("multiply", 5, 3))
print(calculate("divide", 5, 3))
```

```
# calculator_service.py
class CalculatorService:
    def add(self, x, y):
        return x + y

    def subtract(self, x, y):
        return x - y

    def multiply(self, x, y):
        return x * y

    def divide(self, x, y):
        if y == 0:
            return "Error: Division by
zero"
        else:
            return x / y
```

```
# client.py
from calculator_service import
CalculatorService
calculator_service = CalculatorService()

print(calculator_service.add(5, 3))
print(calculator_service.subtract(5, 3))
print(calculator_service.multiply(5, 3))
print(calculator_service.divide(5, 0))
```

LoGrammar

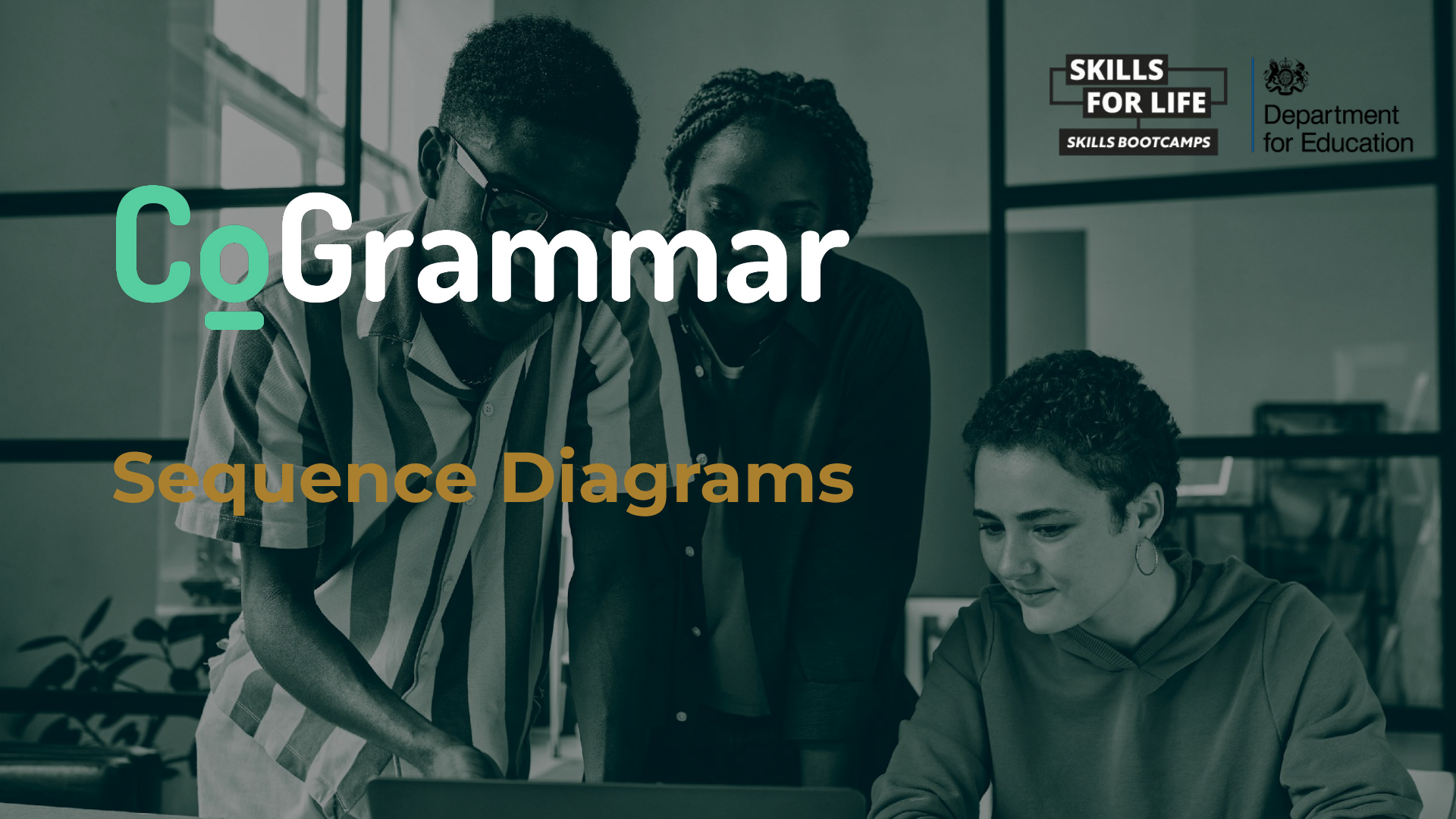
Programming

# Poll

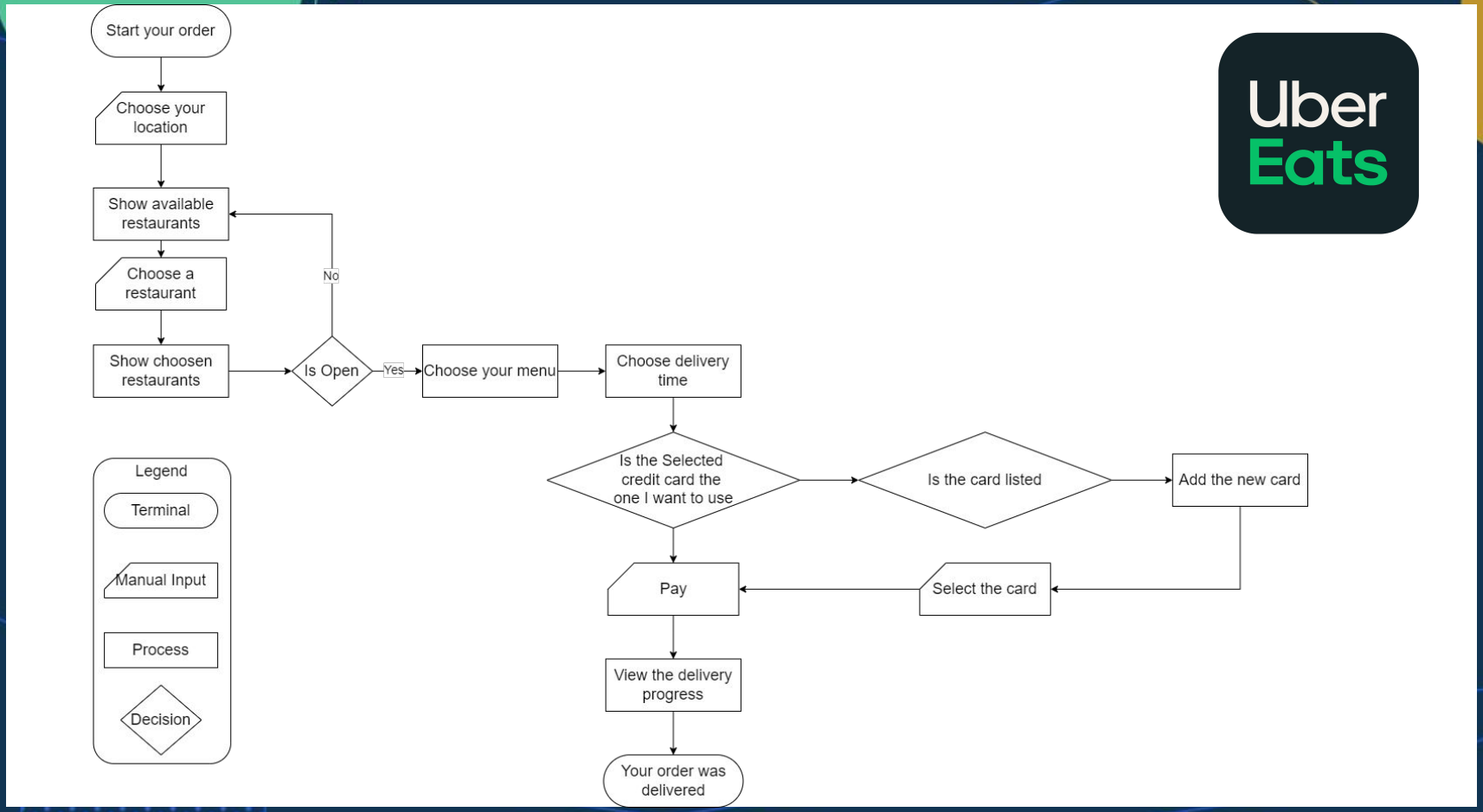
Given the images of the code on the screen, why is one better than the other?

- Lack of modularisation
- Violation of the single responsibility principle
- Lack of readability and maintainability
- Adaptability to new working components
- Everything is in one place
- Testable components

# BREAK!




# Sequence Diagrams




Uber Eats

CoGrammar

Sequence Diagrams

# Intuition

Before diving into the development of a platform like UberEats, it's crucial to map out the sequence of events that occur behind the scenes. Imagine you're craving your favorite meal and open the app to place an order. But what happens next?

How does the app communicate with the restaurant and ensure your food arrives hot and on time? **That's where Sequence diagrams come in.** They help us visualise the **entire journey, from user interaction to backend processing**, ensuring every step is carefully orchestrated for a seamless experience.

# Sequence Diagrams Basics

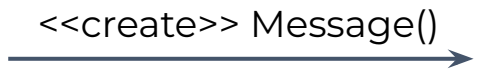
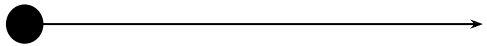
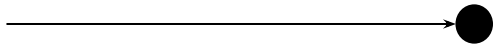
- Shows control flow, the order of interactions
- Time runs vertically, from top to bottom
- Messages run horizontally
- Type of UML diagram

# Poll

Given the definition, what do you think are linked to sequence diagrams ?

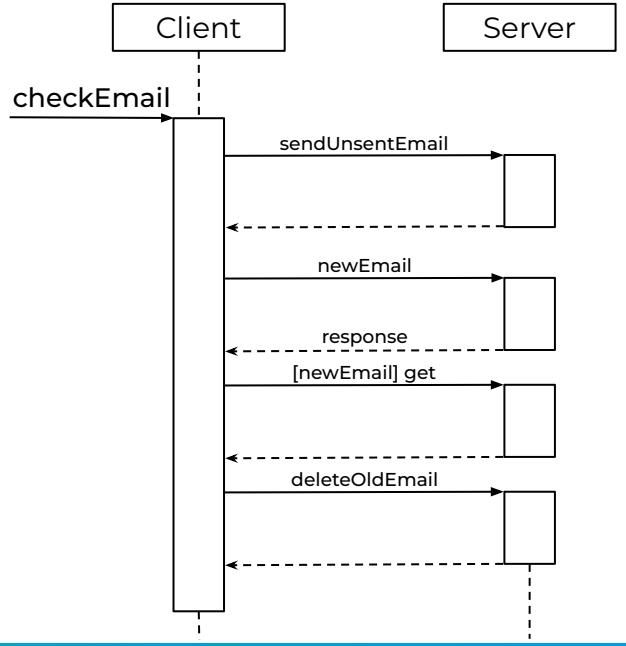
A. Sequence diagram can replace code
B. Sequence diagrams are language-agnostic
C. Representative of the entire software
D. Show relationship between object

# Sequence Diagrams Key Components

| | |
|---|---|
| Synchronous message | |
| Asynchronous message | |
| Message return | |
| Object creation |   |
| Object destruction | |
| Found message |  |
| Lost message |  |

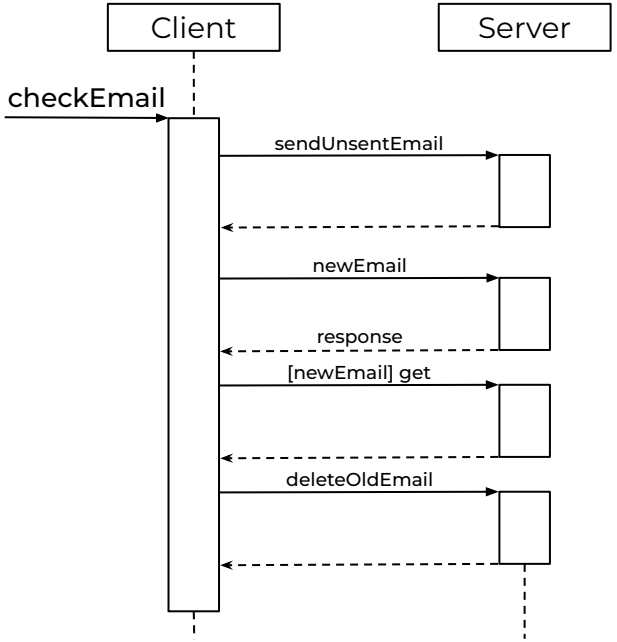
# Sequence Diagram: Managing emails

- **Participant:** an object or an entity; the sequence diagram actor
  - sequence diagram starts with an unattached "found message" arrow



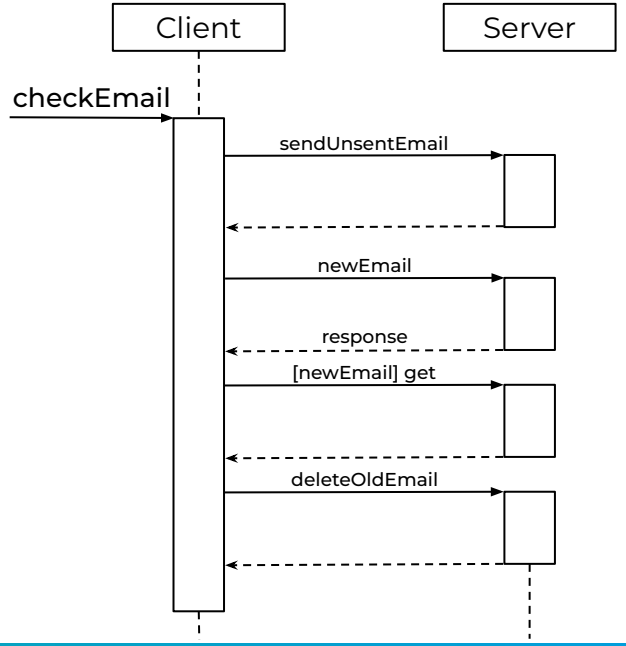
# Sequence Diagram: Managing emails

- **Participant:** an object or an entity; the sequence diagram actor
  - sequence diagram starts with an unattached "found message" arrow
- **Message**: communication between objects

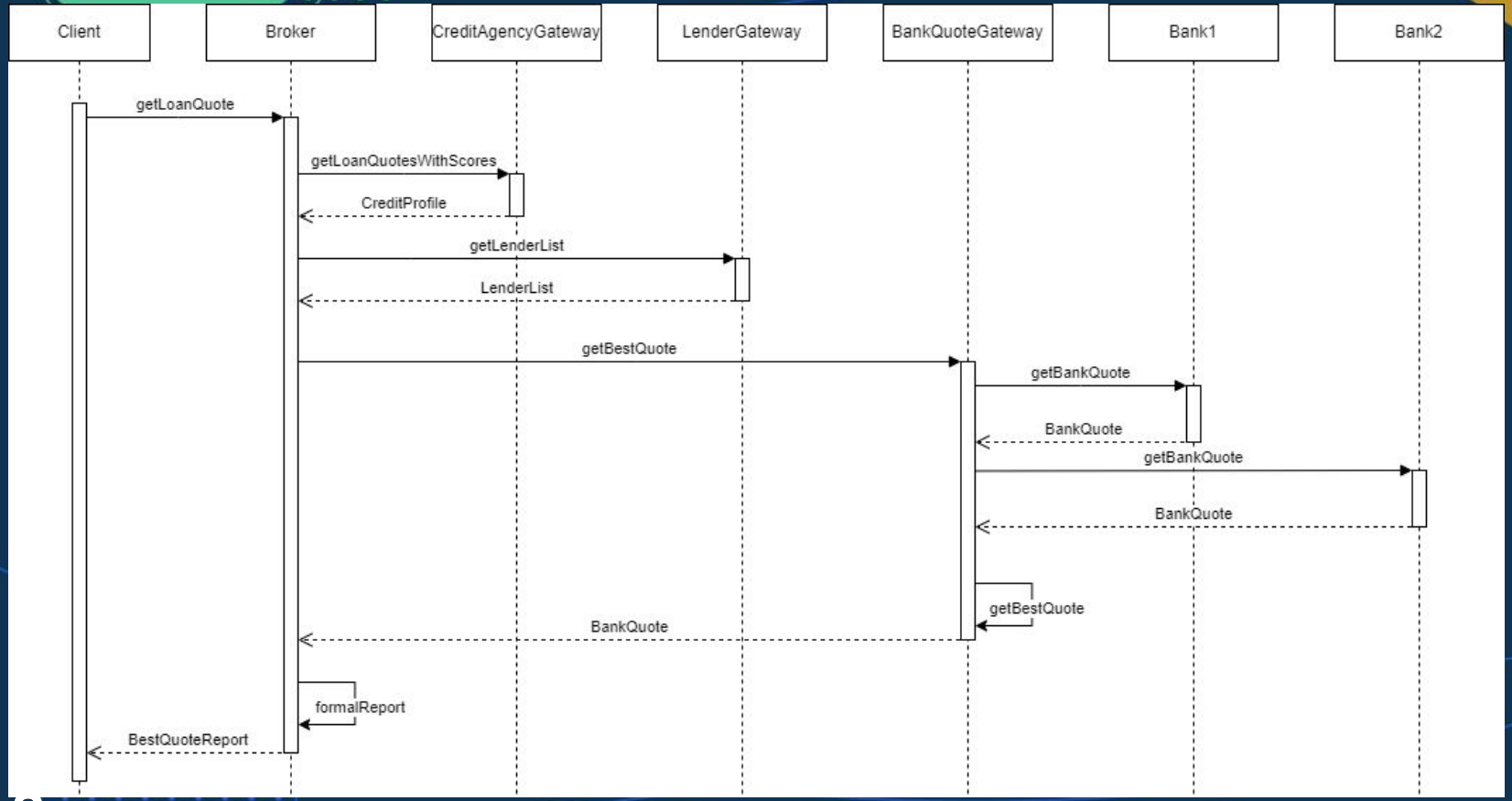


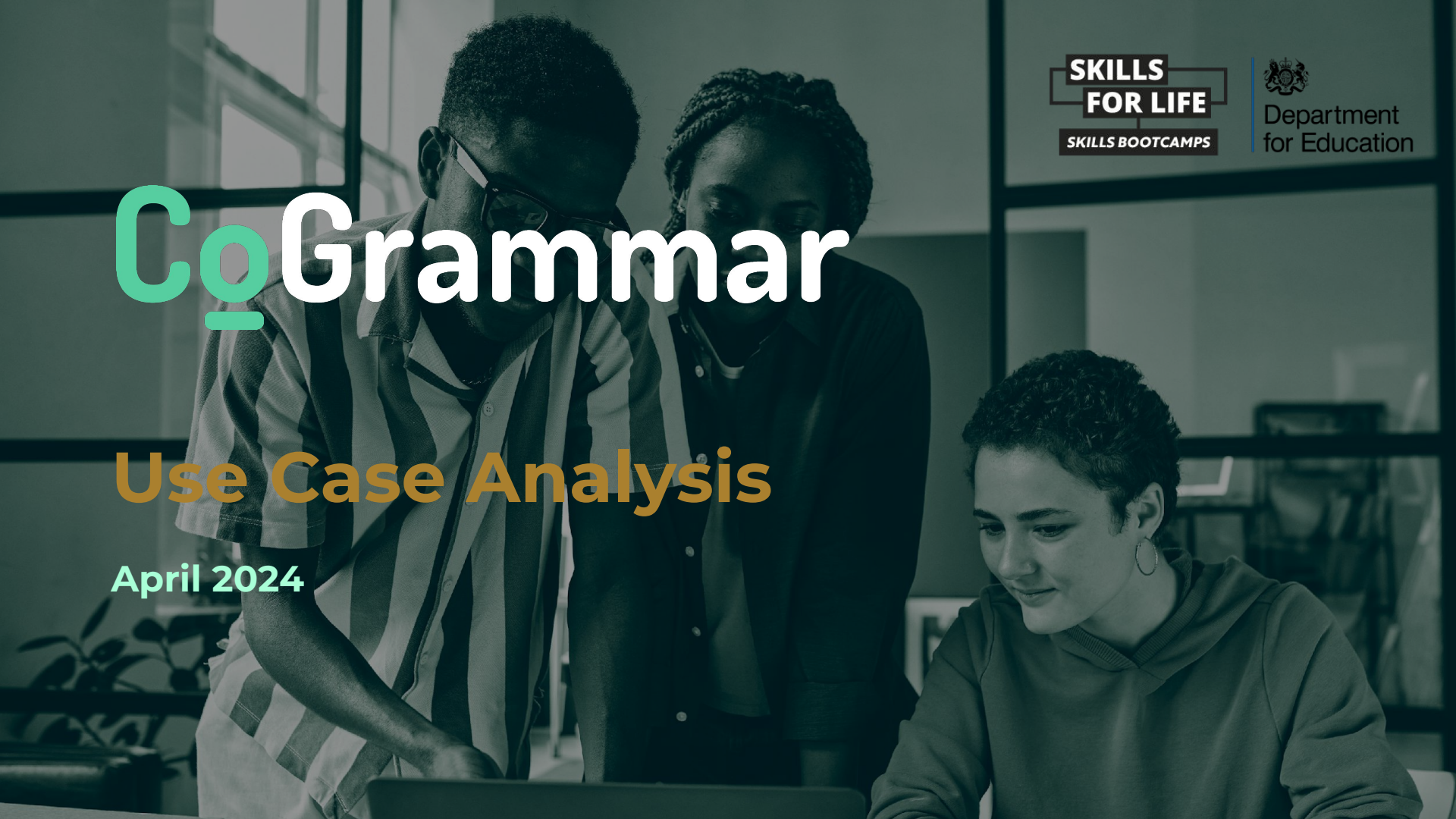
# Sequence Diagram: Managing emails

- **Participant:** an object or an entity; the sequence diagram actor
  - sequence diagram starts with an unattached "found message" arrow
- **Message**: communication between objects
- Axes in a sequence diagram:
  - **horizontal**: which participant is acting
  - **vertical**: time (↓ forward in time)



# Use Case: Loan Application






# Use Case Analysis

April 2024

# Intuition

While the loan system may appear straightforward to users, behind the scenes, a complex process unfolds involving multiple stakeholders and systems. **Use case diagrams offer a comprehensive overview of this process, illustrating the interactions between actors like customers, loan officers, and administrators, and the system itself.**

By visualizing the various scenarios and functionalities required for loan processing, these diagrams provide a clear understanding of the system's core capabilities and how different users can interact with it to achieve their objectives effectively.

# Use Case Diagrams Basics

- Describe functionality from the user's perspective
- One (or more) use-cases per kind of user
    - May be many kinds in a complex system
- Use-cases capture requirements
- Type of UML diagram

# Poll

In the context of a bank's loan provision system, which statements about use case diagrams are true?

A. Use case diagrams visualise interactions between actors and the loan system.

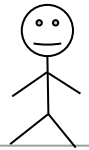
B. Use case diagrams depict the sequence of events in loan processing.

C. Use case diagrams represent the internal structure of the loan system.

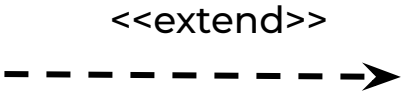
D. Use case diagrams show the implementation details of the loan system.

E. Use case diagrams are used primarily for designing graphical user interfaces

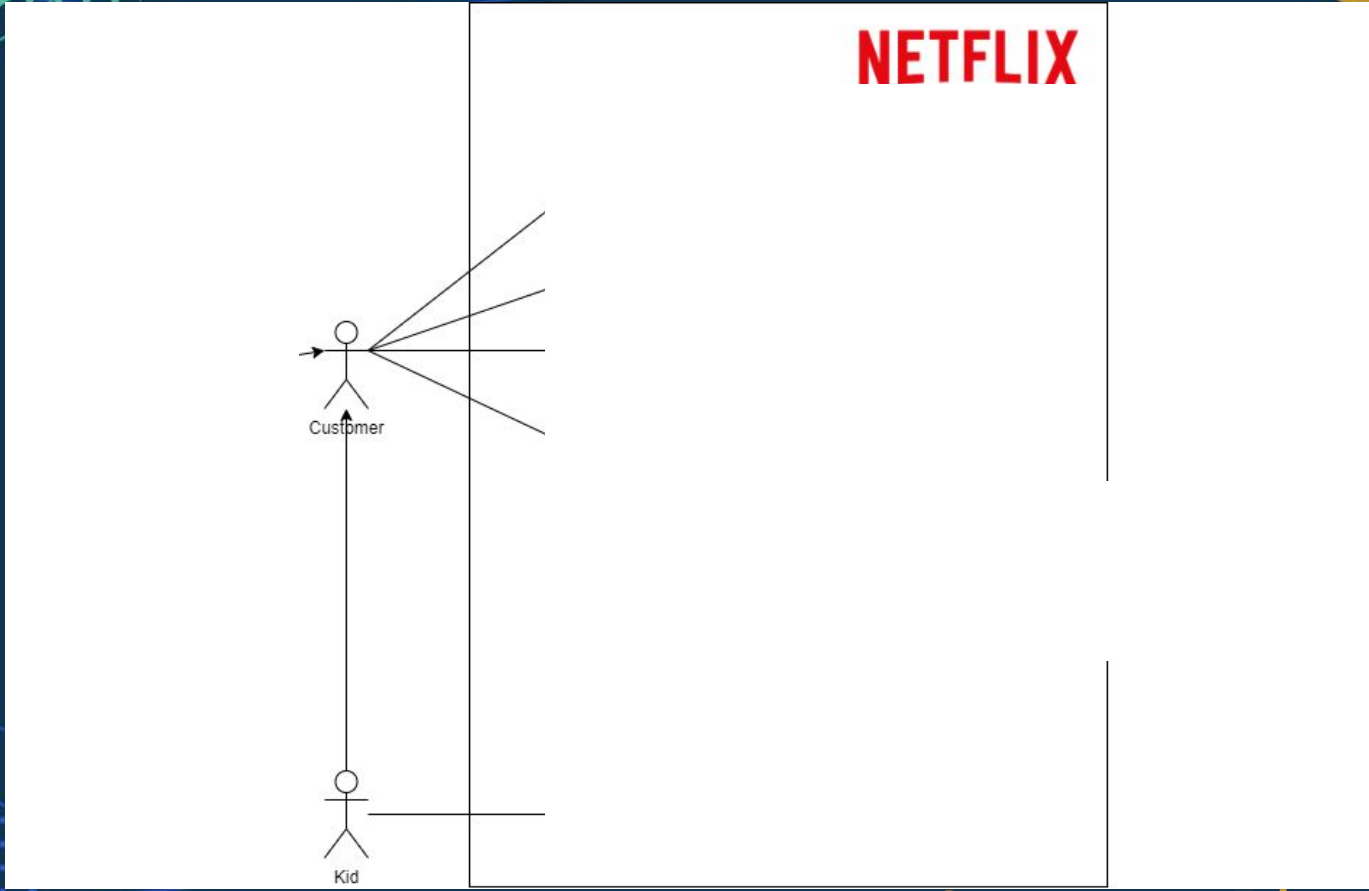
# Use Case Diagrams Key Components

| | | |
|---|---|---|
| Actor |  | Human or system interacting with the system. People, Devices, External Systems |
| Use Cases | | Documents the system behaviour from the actor's point of view. |
| System | | Helps identify what is external versus internal, and what the responsibilities of the system are. |

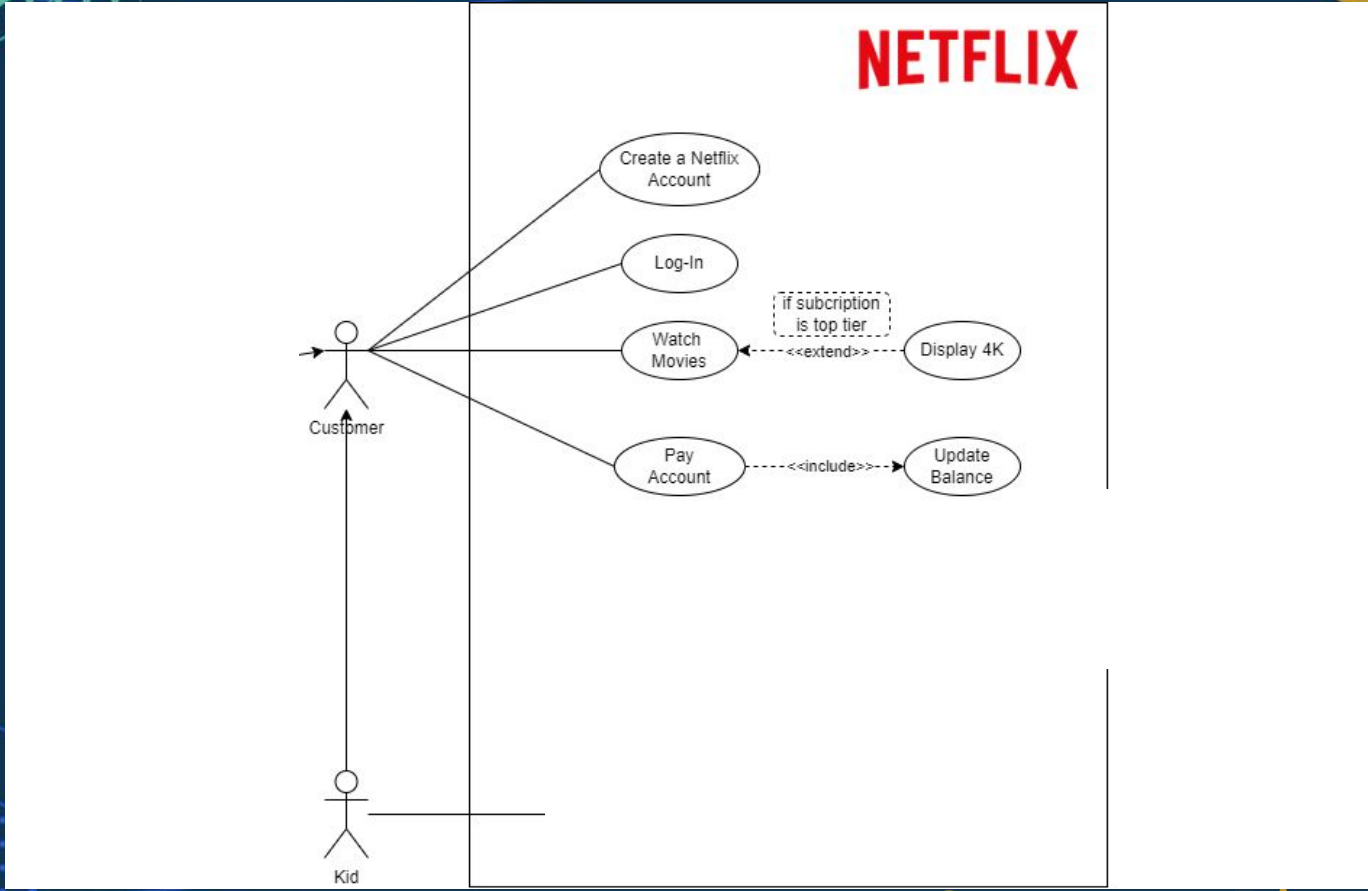
# Use Case Diagrams Key Components

| | | |
|---|---|---|
| Association | | A actor must be associated with at least one use case<br>Multiple actors can be associated with one use case |
| Extend |  | To extend the functionality of a use case, given a condition. |
| Generalization | | An actor can inherit the role of another one |
| Uses | | When a use case uses another process |
| Include | | SHow the included or implicit behaviour of a use case |

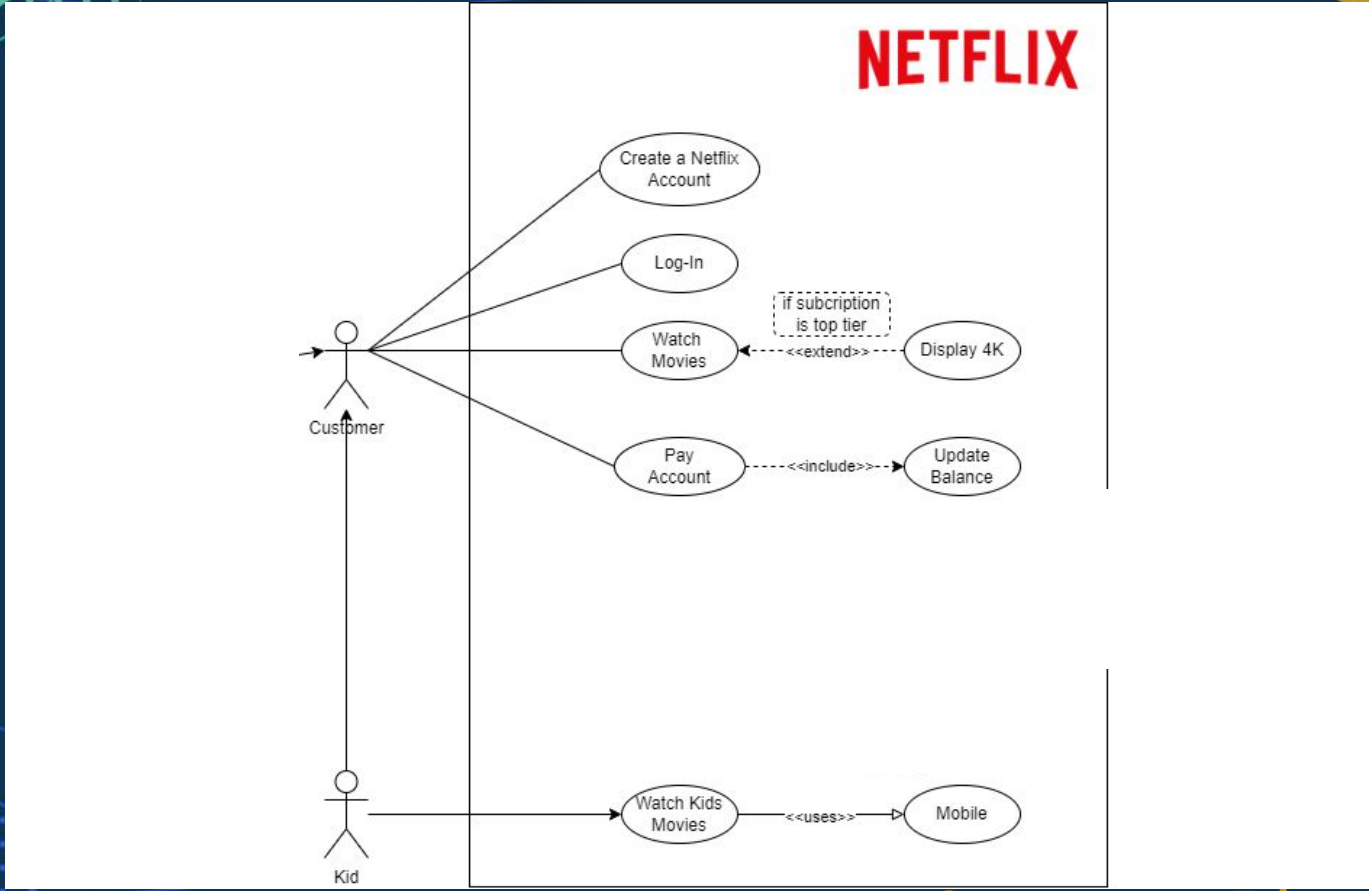
# Use Case: Netflix: Video Streaming



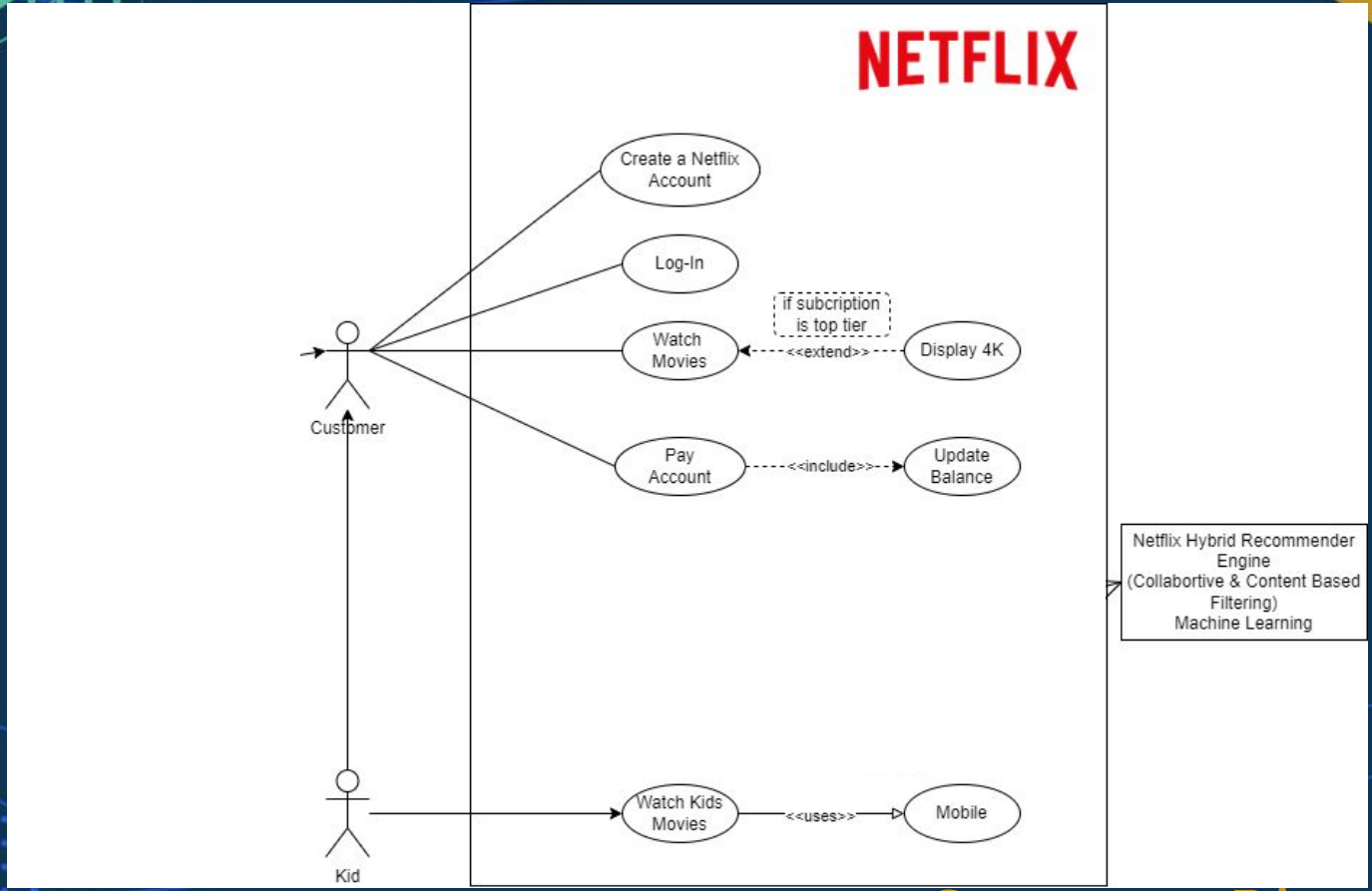
# Use Case: Netflix: Video Streaming



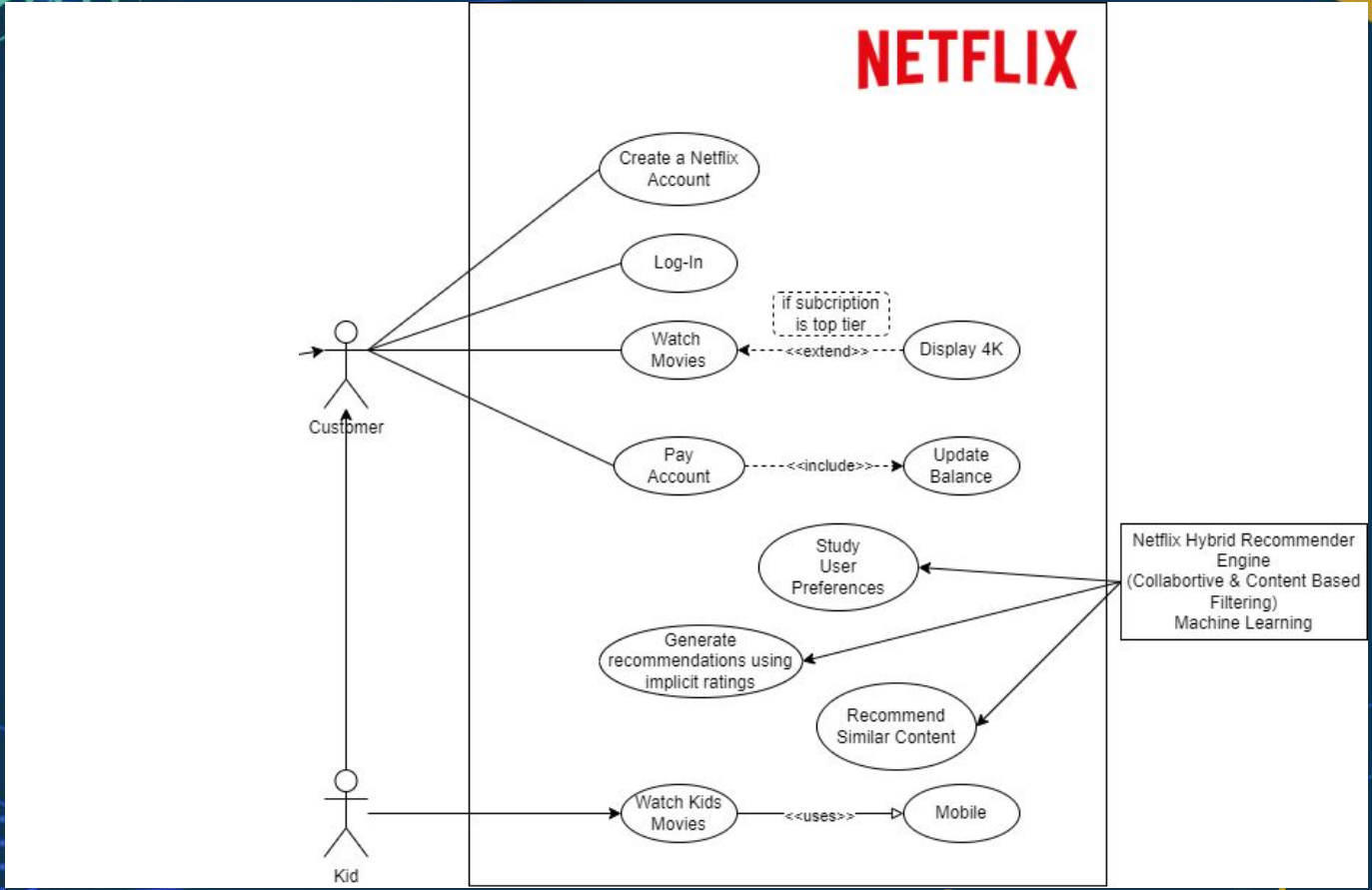
# Use Case: Netflix: Video Streaming



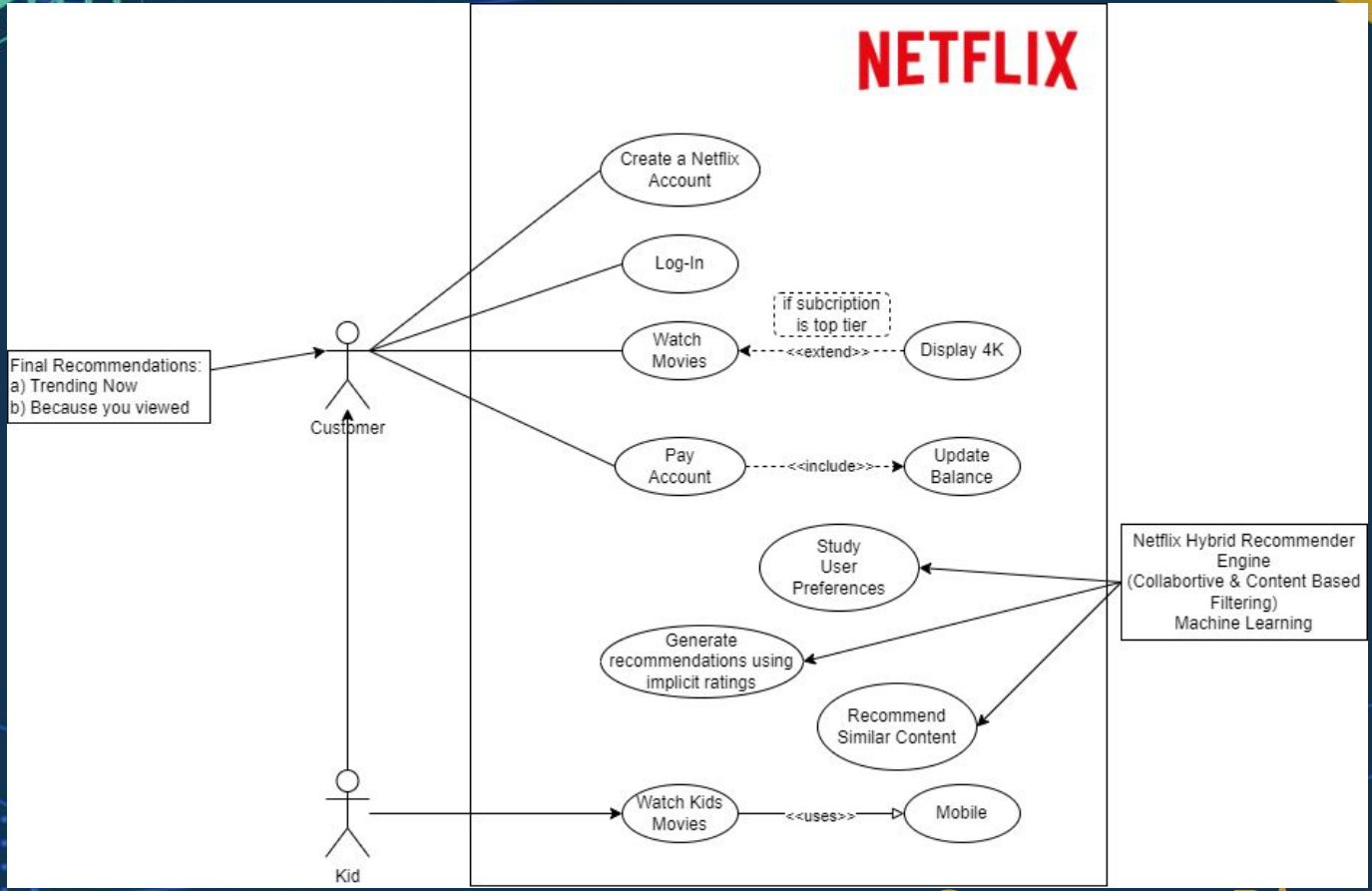
# Use Case: Netflix: Video Streaming



# Use Case: Netflix: Video Streaming



# Use Case: Netflix: Video Streaming



# Summary

- **modularisation:** Breaking down software into independent modules enhances maintainability, scalability, and reusability.
- **Sequence Diagrams:** Visualising component interactions elucidates system behaviour over time, aiding in comprehension and optimization.
- **Use Case Diagrams:** Representing system functionalities from user viewpoints aids in requirement analysis and stakeholder communication.
- **modularisation Importance:** It streamlines development, promotes code reusability, and enables easier collaboration among teams.
- **Sequence Diagrams' Utility:** They highlight the sequence of events, showcasing message exchanges and system flow.
- **Use Case Diagrams' Significance:** They provide a holistic view of system functionalities, ensuring alignment with user needs and expectations.
- **Overall Importance:** Together, these tools facilitate the design, development, and understanding of complex software systems, ensuring efficiency and user satisfaction.

# Follow-up Activities

Think about doing such diagrams for an e-Commerce system.

Use all the concepts we've covered.

# Thank you for attending

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar