Welcome to the

CoGrammar

Use Case Analysis and Sequence Diagrams

The session will start shortly...

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

CoGrammar

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** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- 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

Skills Bootcamp
8-Week Progression Overview

Fulfil 4 Criteria to Graduation

✅ **Criterion 1: Initial Requirements**

- ***Guided Learning Hours (GLH):*** Minimum of 15 hours
- ***Task Completion:*** First 4 tasks

Due Date: 24 March 2024

✅ **Criterion 2: Mid-Course Progress**

- ***Guided Learning Hours (GLH):*** Minimum of 60 hours
- ***Task Completion:*** *First* 13 tasks

Due Date: 28 April 2024

CoGrammar

Skills Bootcamp Progression Overview

✅ Criterion 3: Course Progress

- ***Completion:*** All mandatory tasks, including Build Your Brand and resubmissions by study period end
- ***Interview Invitation:*** Within 4 weeks post-course
- ***Guided Learning Hours:*** Minimum of 112 hours by support end date (10.5 hours average, each week)

✅ Criterion 4: Demonstrating Employability

- ***Final Job or Apprenticeship Outcome:*** Document within 12 weeks post-graduation
- ***Relevance:*** Progression to employment or related opportunity

CoGrammar

Learning Objectives & Outcomes

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

CoGrammar

SKILLS FOR LIFE
SKILLS BOOTCAMPS

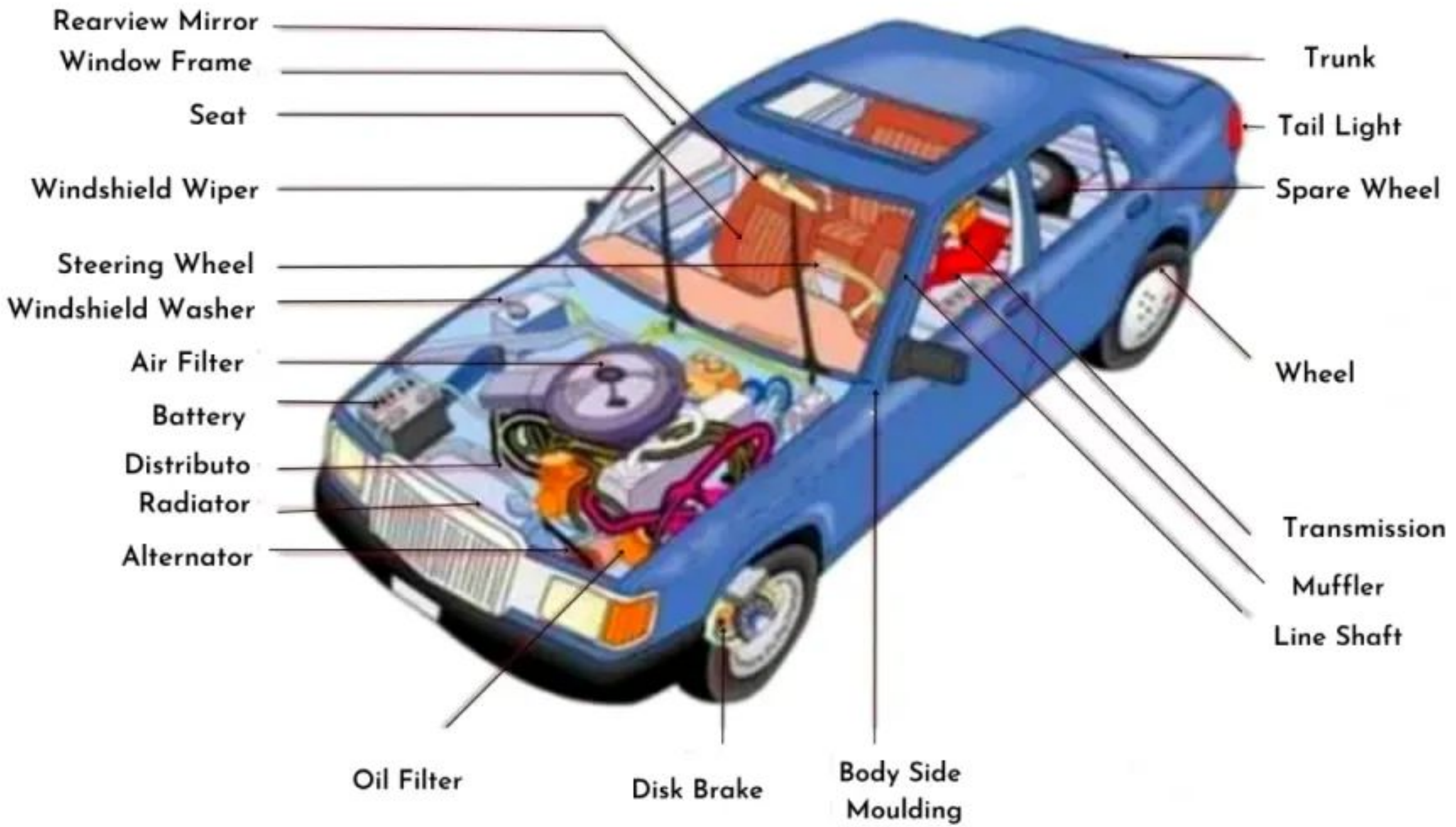
Department for Education

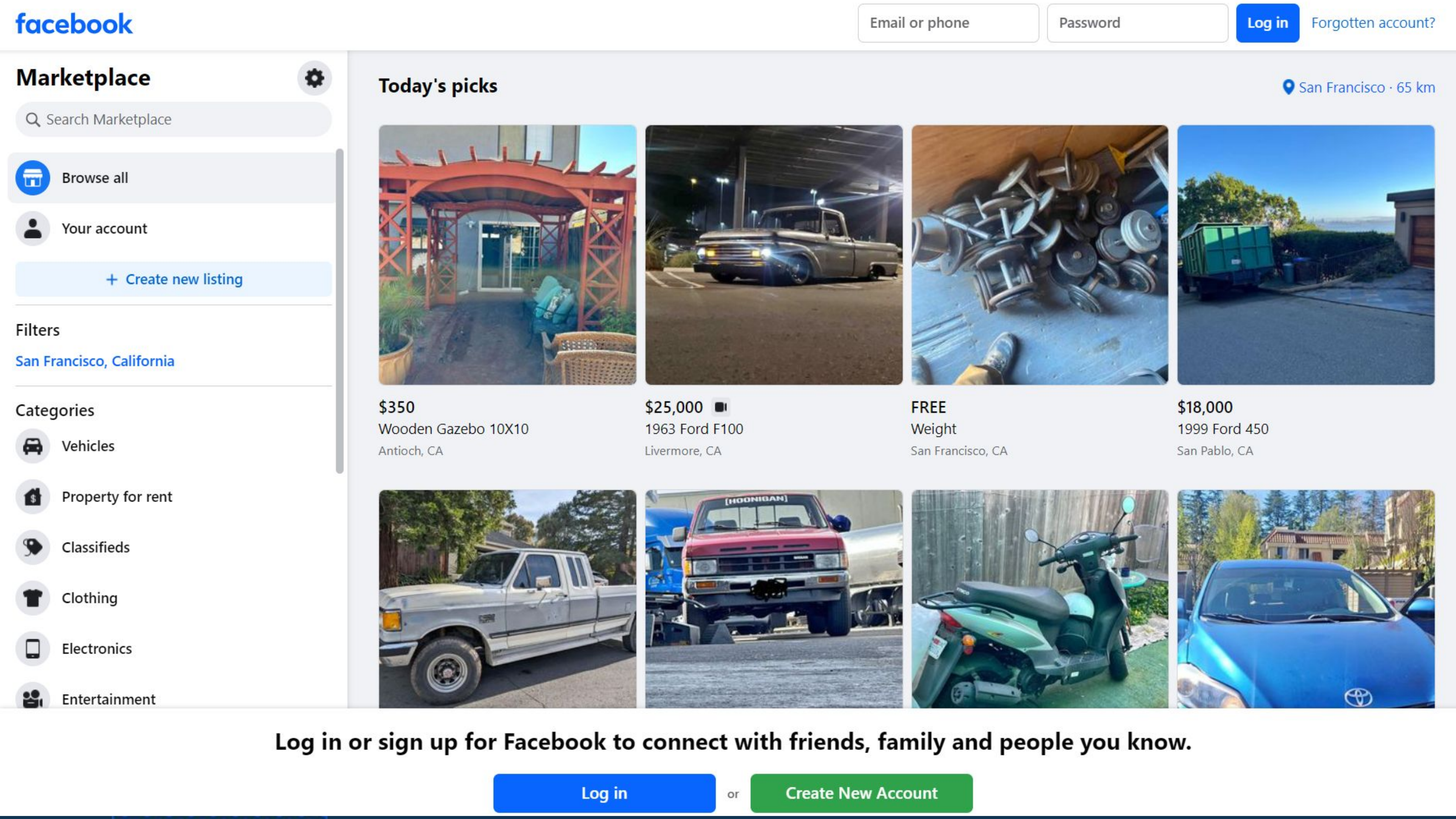
CoGrammar

Modular Programming

April 2024

PARTS OF A CAR





Marketplace

Filters

San Francisco, California

Categories

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.

Intuition Poll

- Given the image of Facebook Marketplace and a vehicle, what are the similarities between them?
    - Maintainable components
    - Reusable components
    - Monolithic design
    - Adaptability to new working components
    - Coupled components
    - 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.

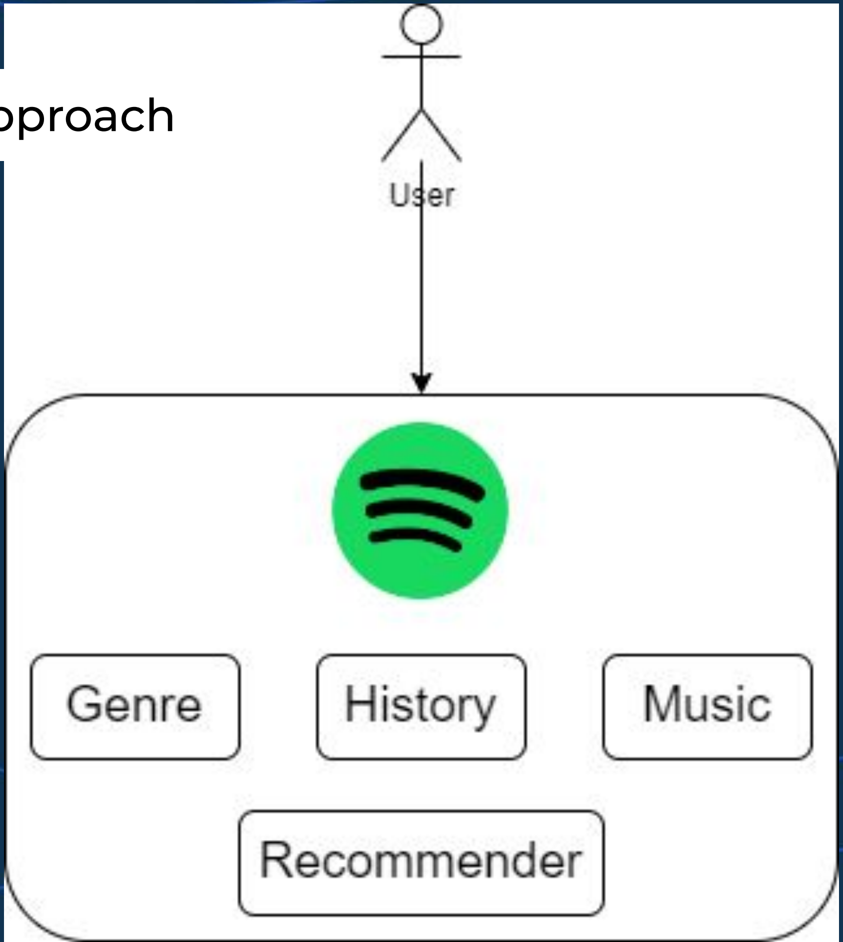
Definition and Importance

- 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.

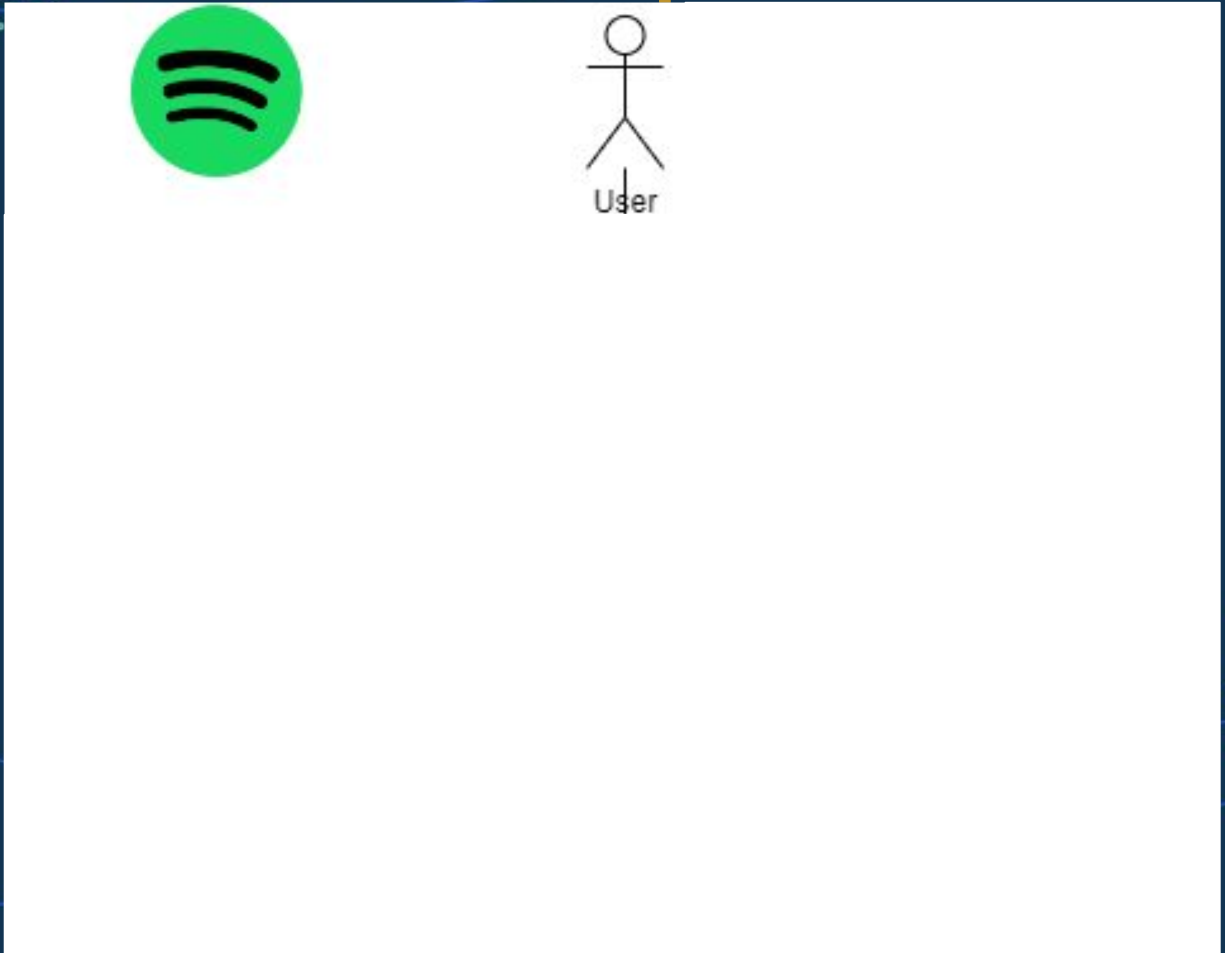
Definition and Importance

Monolithic approach



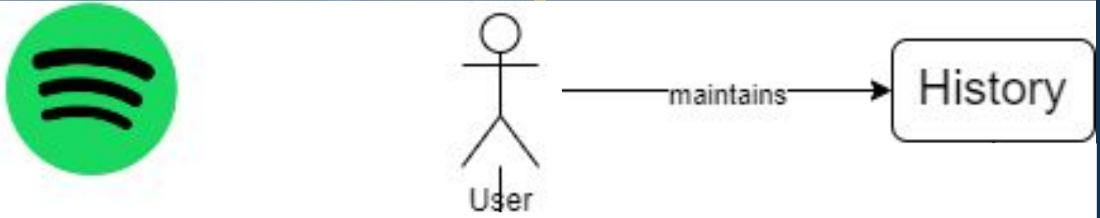
CoGrammar

Definition and Importance



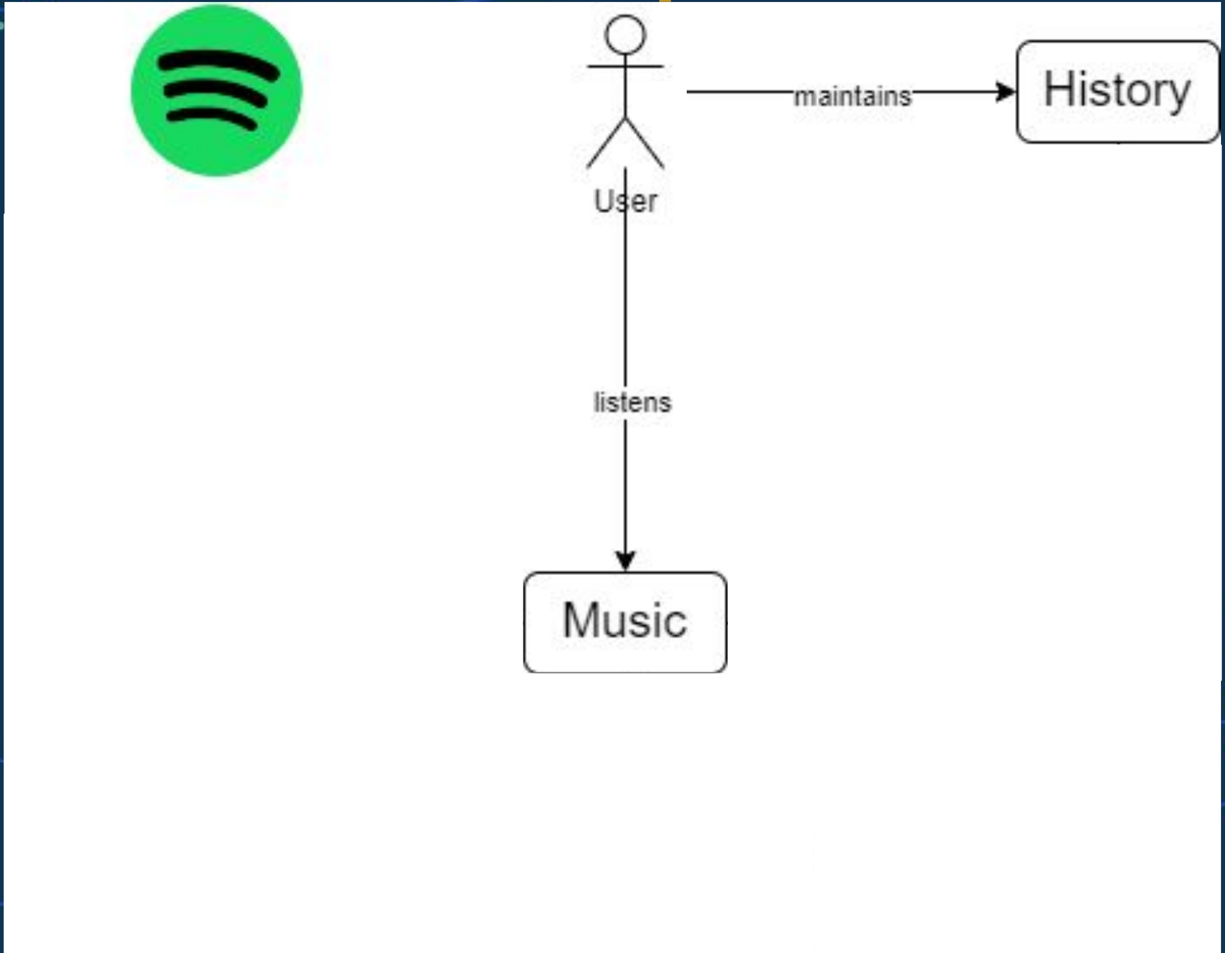
CoGrammar

Definition and Importance



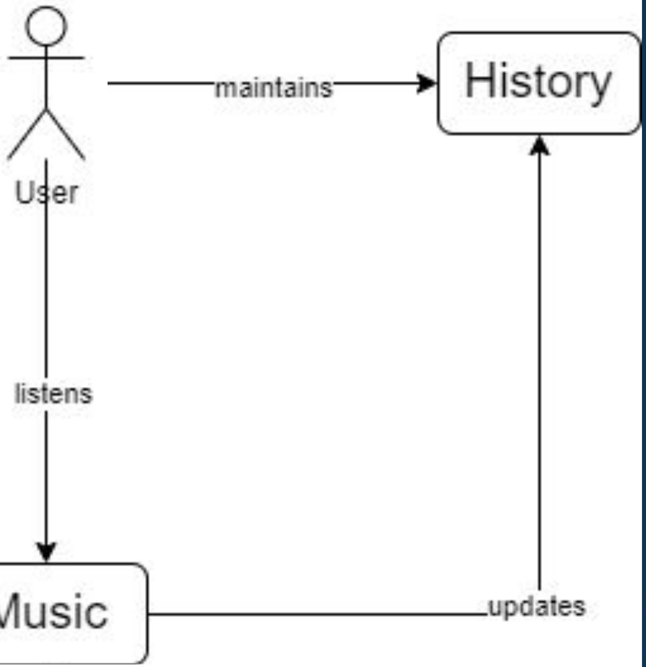
CoGrammar

Definition and Importance



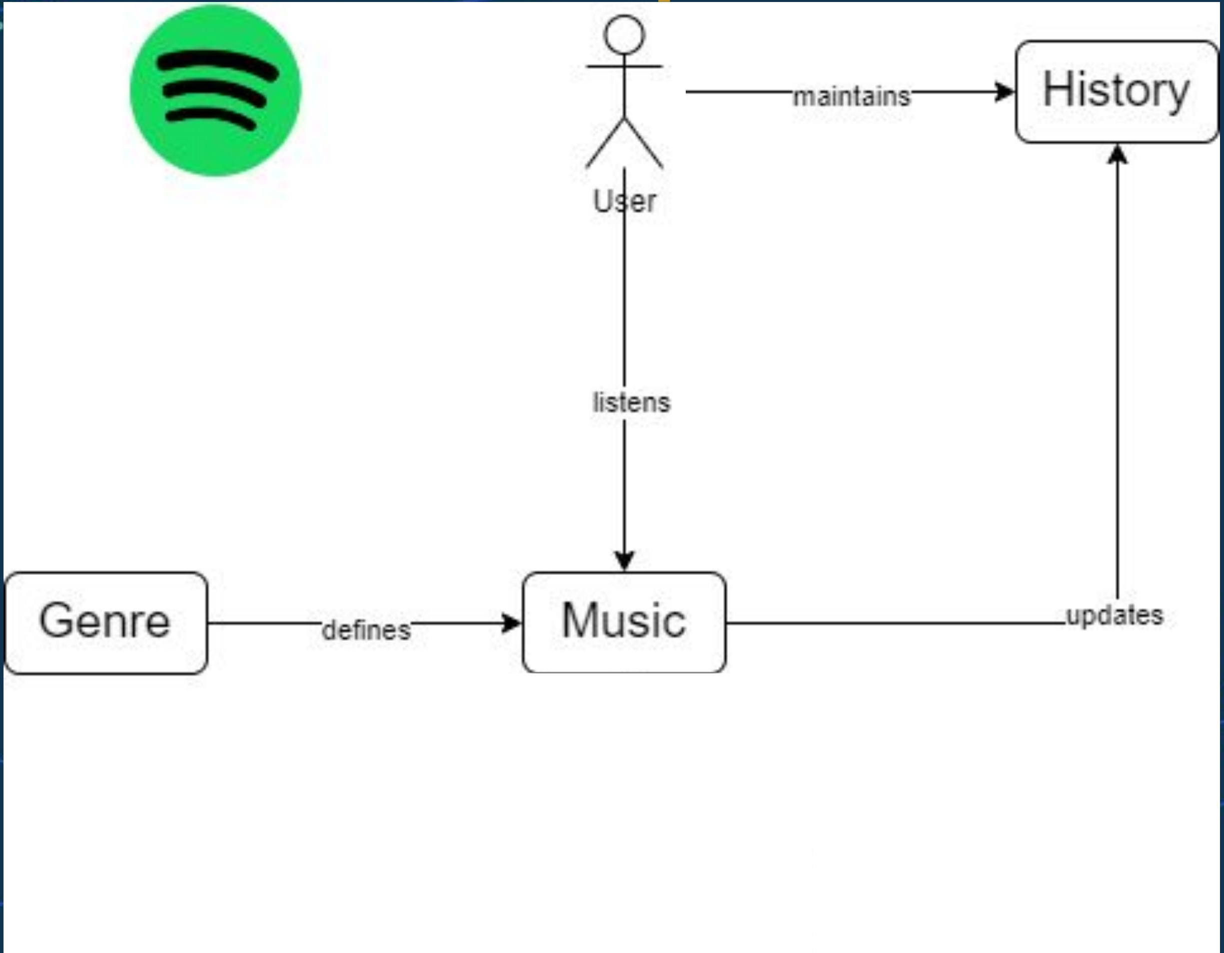
CoGrammar

Definition and Importance



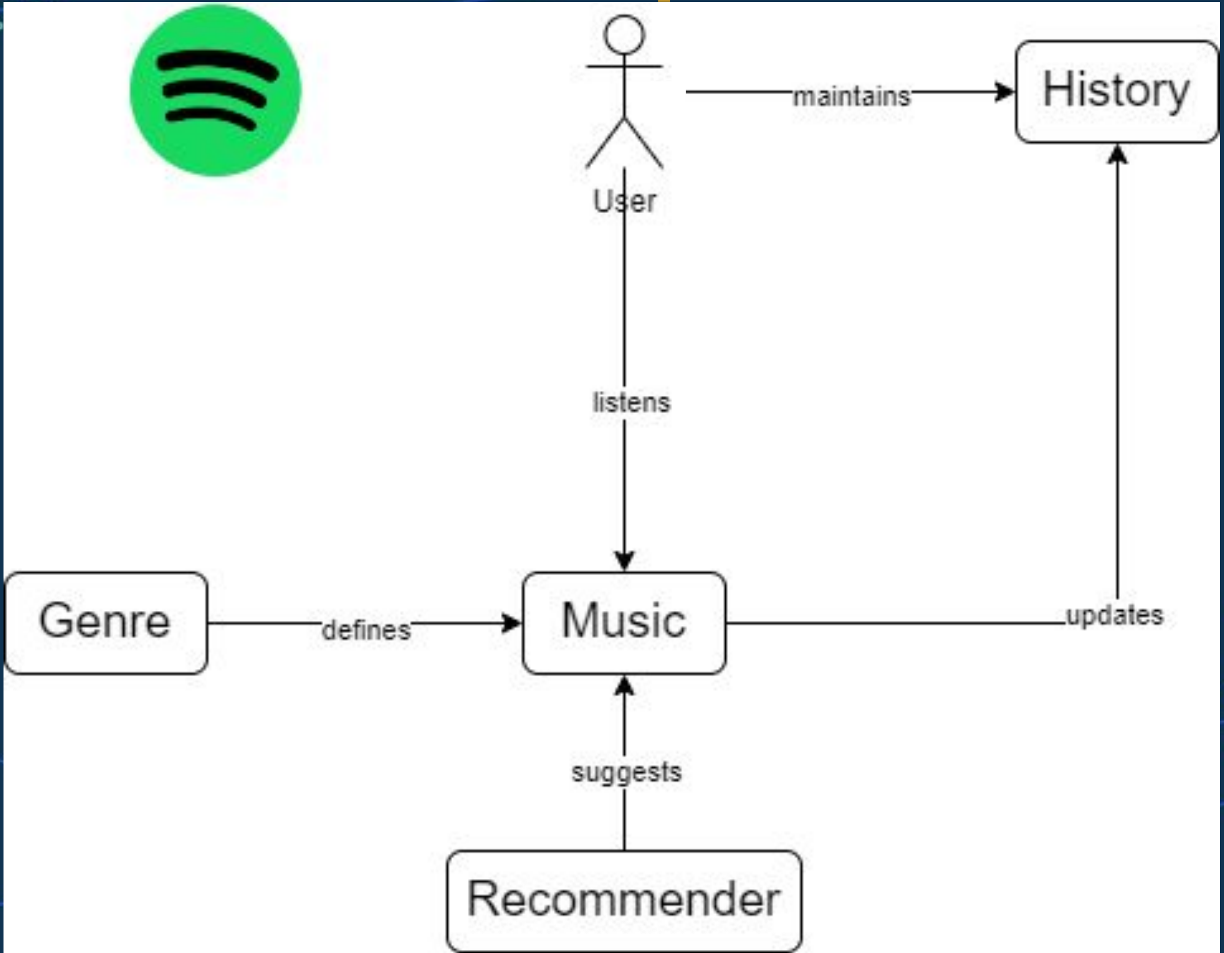
CoGrammar

Definition and Importance



CoGrammar

Definition and Importance



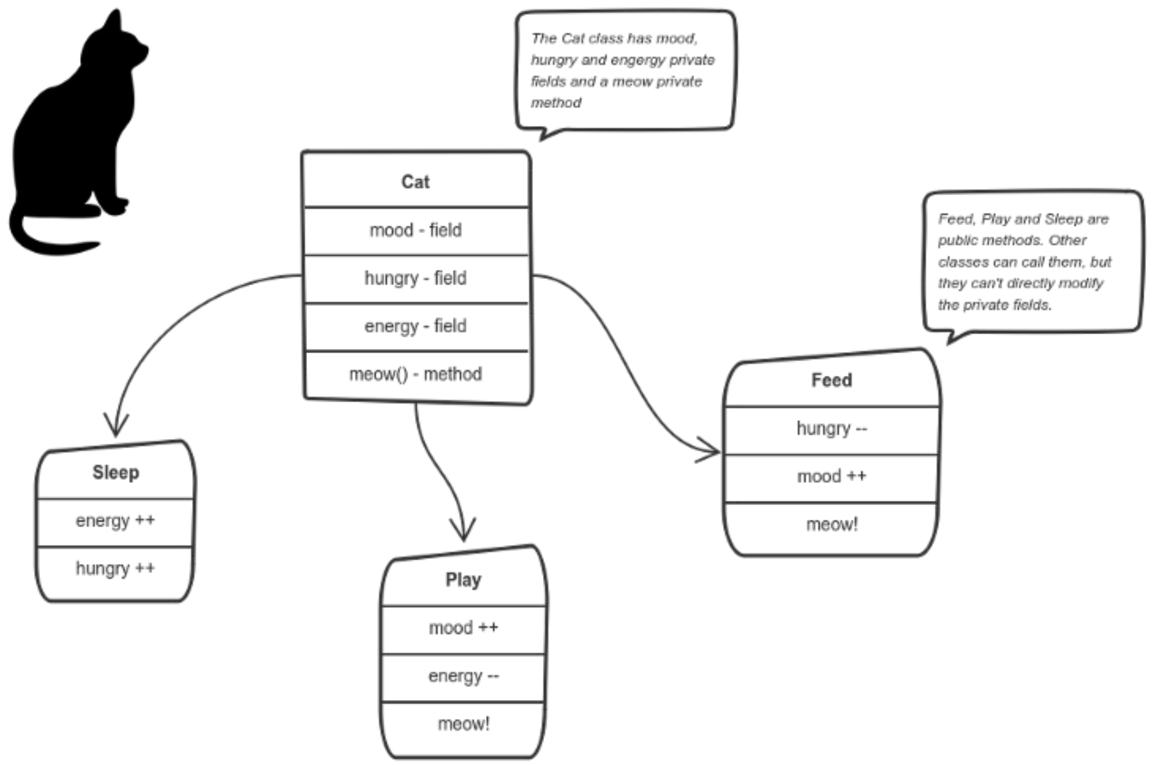
CoGrammar

Techniques and Approaches

- This design principle also enhances portability, as modules can be transferred across different environments or platforms with minimal adjustments.

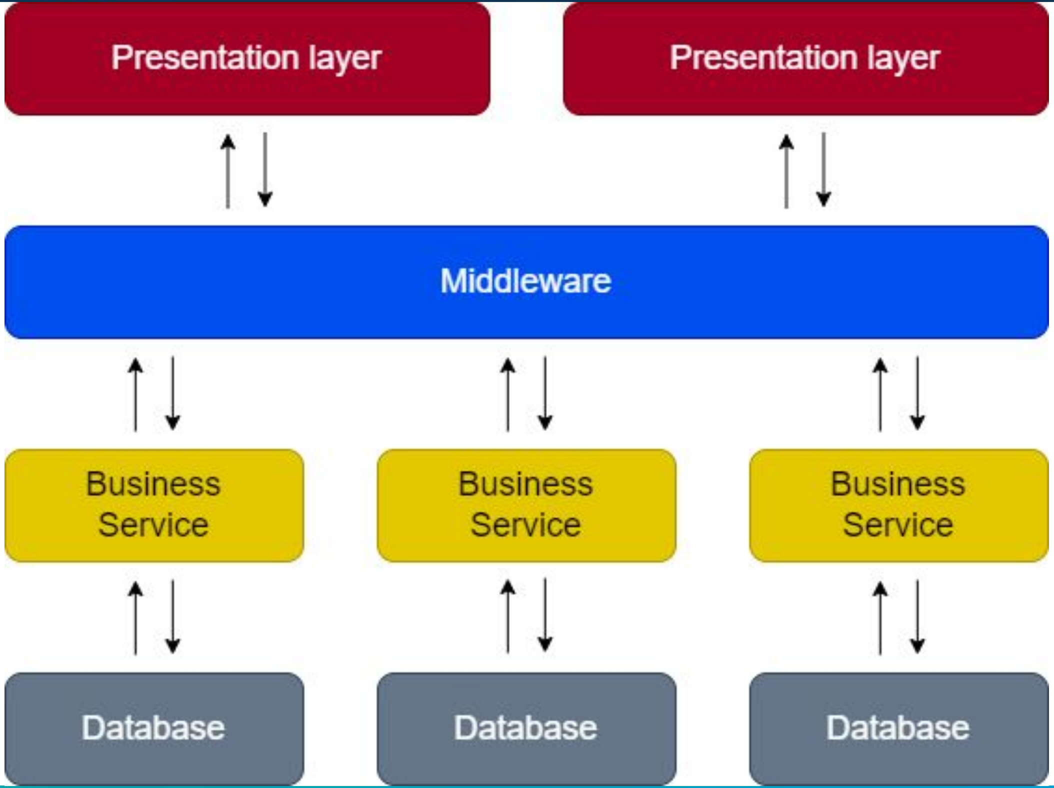
- Modularity promotes maintainability by isolating changes and facilitating debugging and updates, ultimately improving the robustness and flexibility of software systems.

Object Oriented Programming



The Cat class has mood, hungry and engergy private fields and a meow private method
Cat
mood - field
hungry - field
energy - field
meow() - method
Feed, Play and Sleep are public methods. Other classes can call them, but they can't directly modify the private fields.
Sleep
energy ++
hungry ++
Play
mood ++
energy --
meow!
Feed
hungry --
mood ++
meow!

Service Oriented Architecture (SOA)



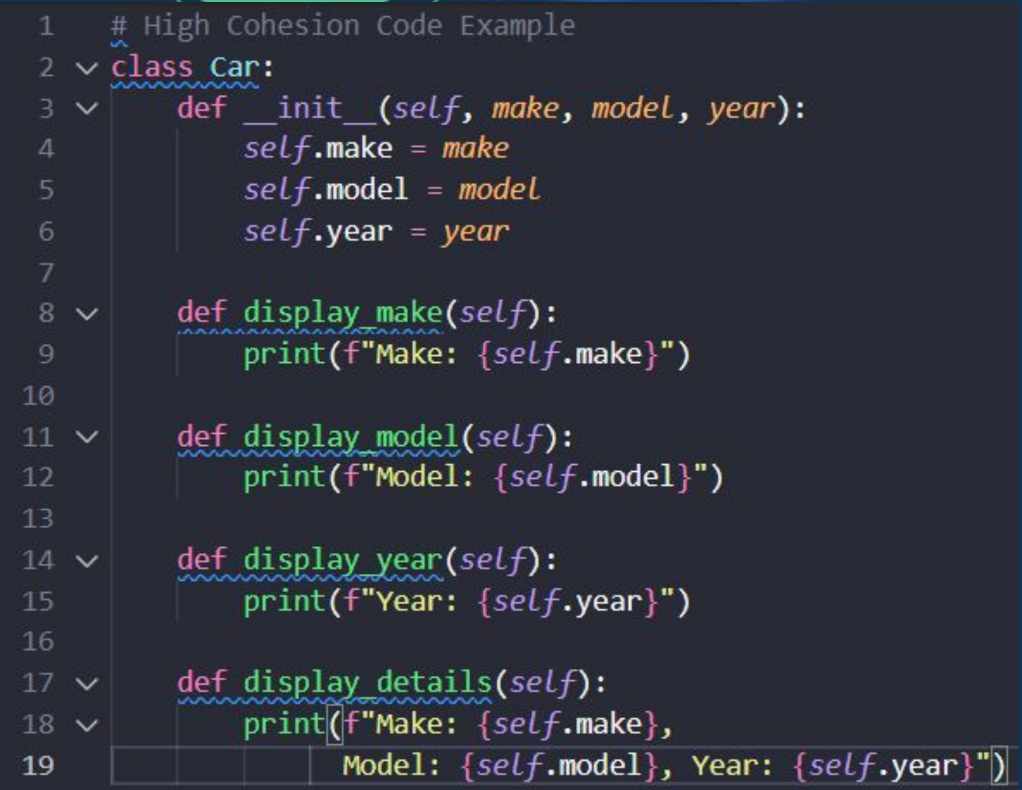
Cohesion and Coupling

Coupling describes the relationships between modules, and cohesion describes the relationships within them.

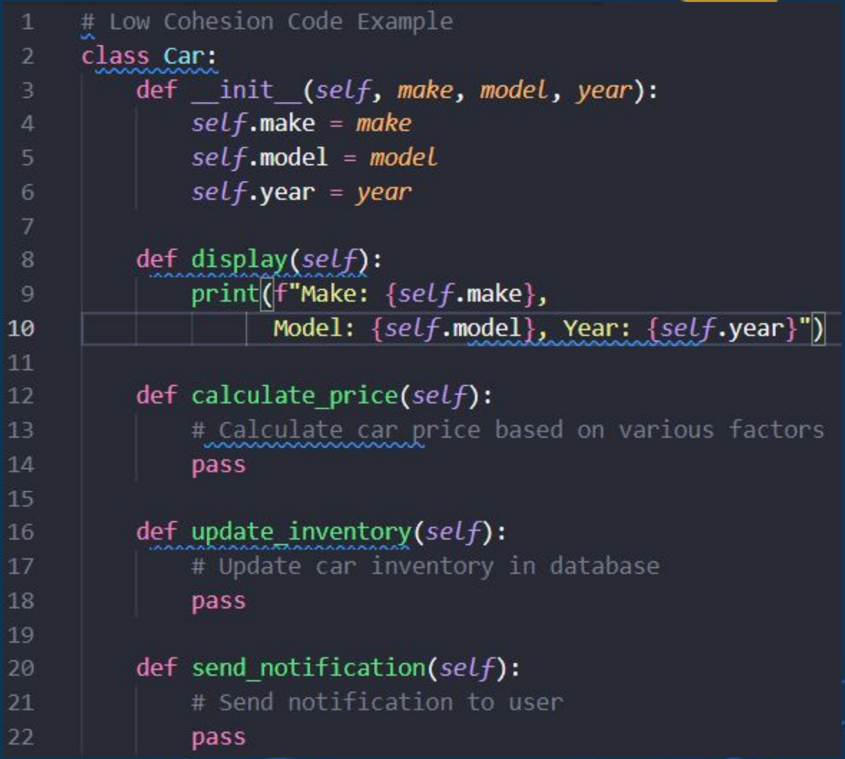
Coupling is the measure of the degree of interdependence between the modules. In a good design, the various component parts (e.g. the classes) have low coupling.

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

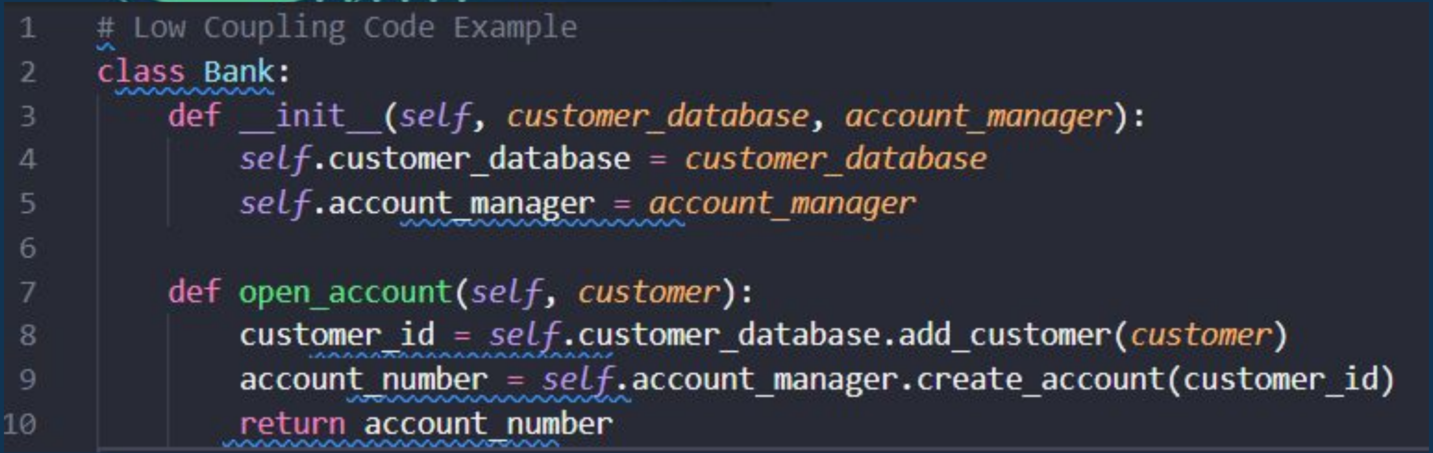


```python
# High Cohesion Code Example
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    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}")
```

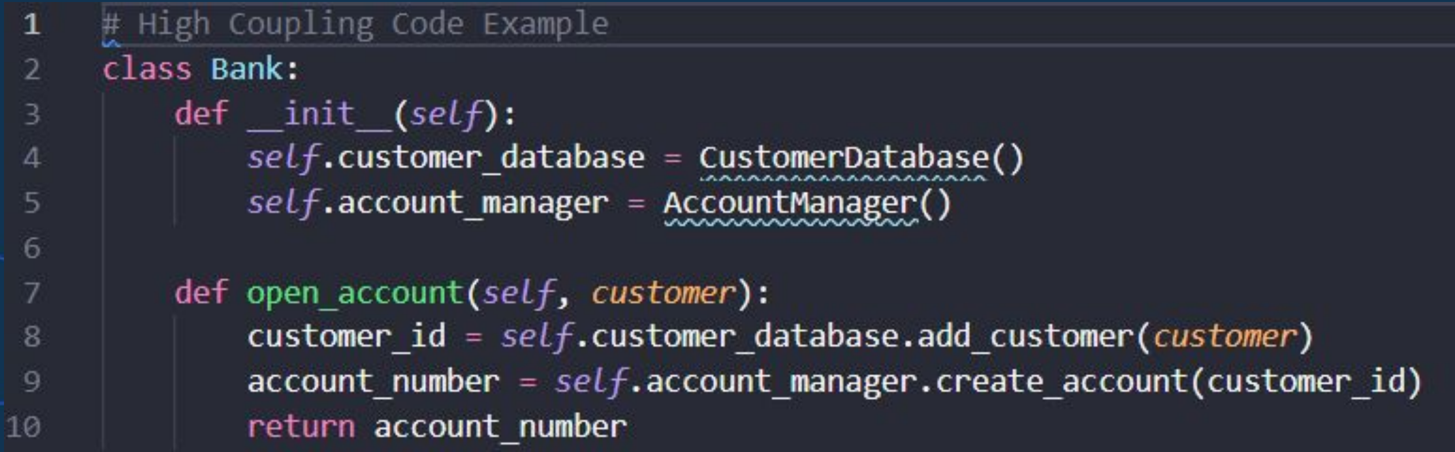


```python
# Low Cohesion Code Example
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

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

    def calculate_price(self):
        # Calculate car price based on various factors
        pass

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

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

Coupling



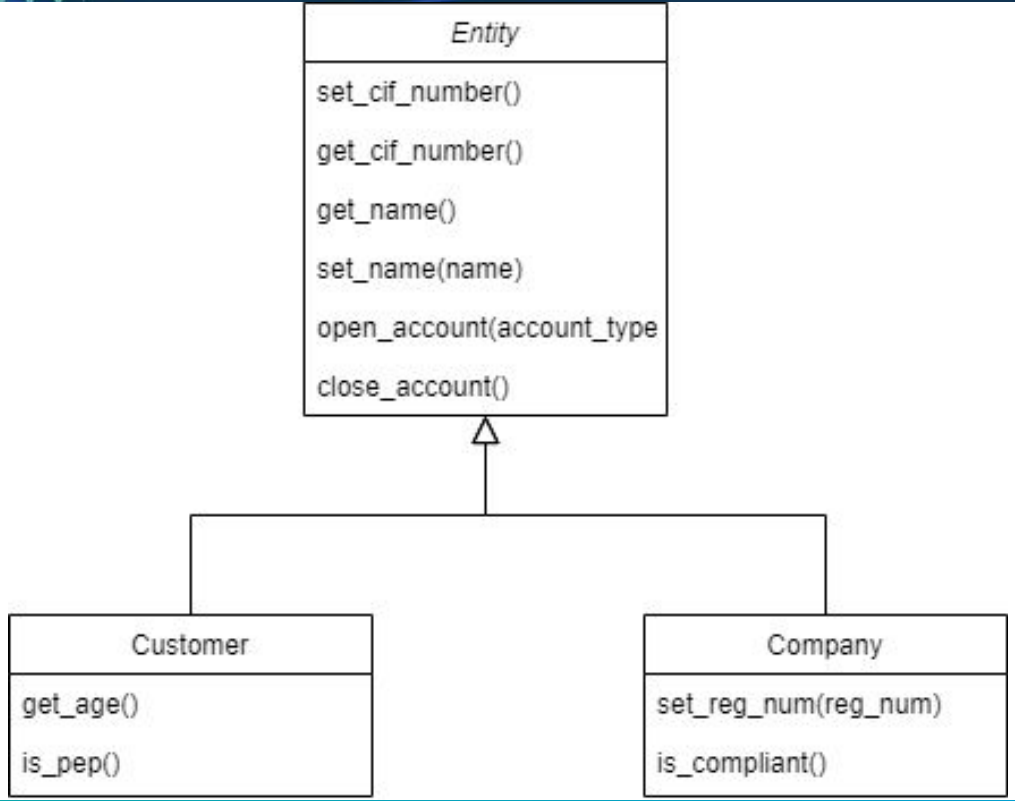
```python
# Low Coupling Code Example
class Bank:
    def __init__(self, customer_database, account_manager):
        self.customer_database = customer_database
        self.account_manager = account_manager

    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
```

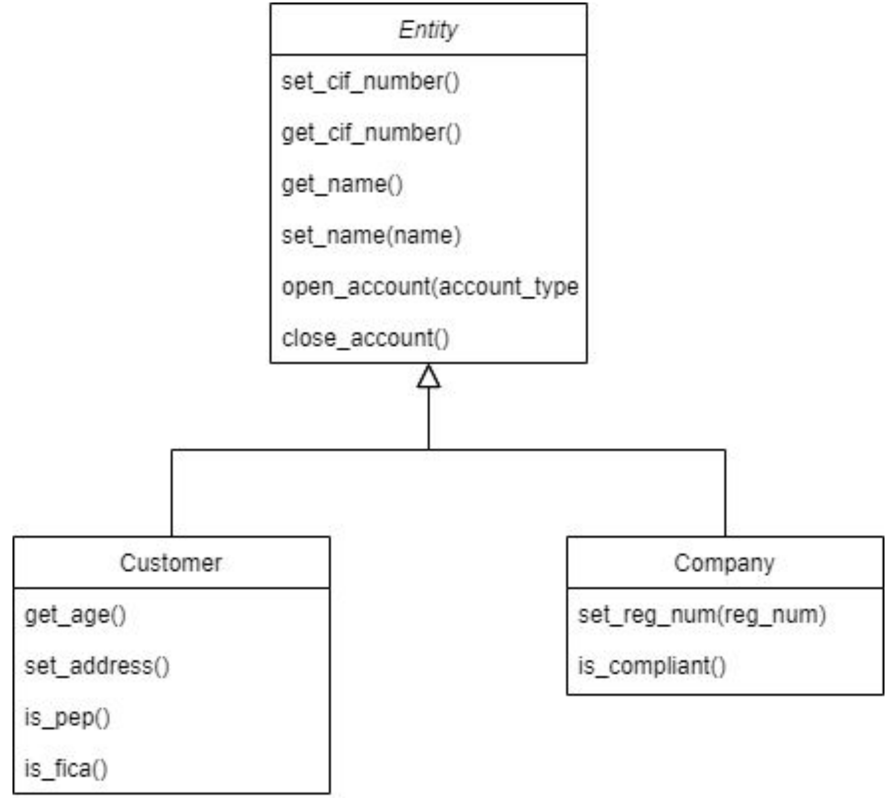
CoGrammar

Open/Closed Principle



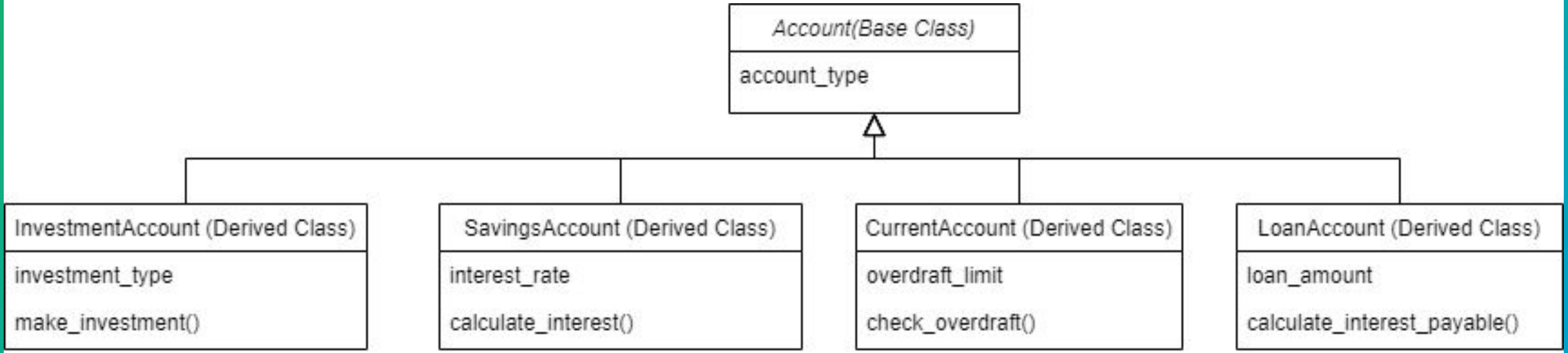
CoGrammar

Open/Closed Principle



CoGrammar

Open/Closed Principle


Account(Base Class)
account_type
InvestmentAccount (Derived Class)
investment_type
make_investment()
SavingsAccount (Derived Class)
interest_rate
calculate_interest()
CurrentAccount (Derived Class)
overdraft_limit
check_overdraft()
LoanAccount (Derived Class)
loan_amount
calculate_interest_payable()

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.

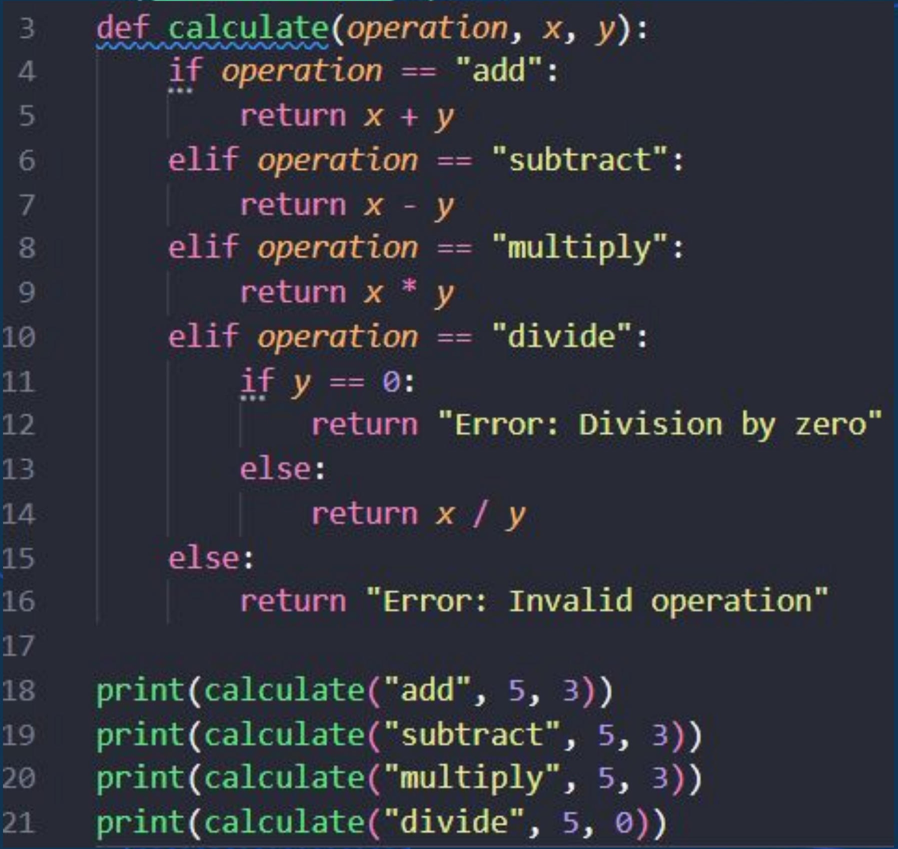
Benefits of Modularisation

- **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.

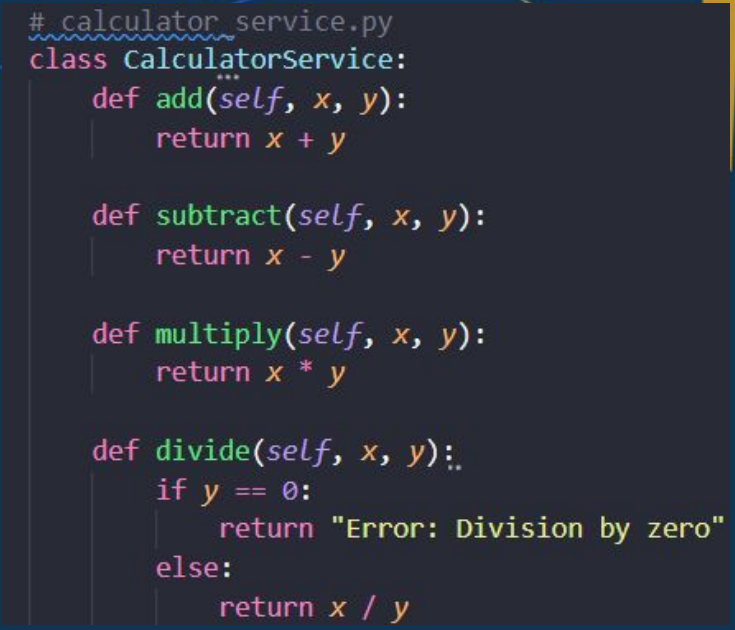
- **Flexibility:** Modular programming enables developers to modify or replace individual modules without impacting other parts of the system. This flexibility allows for easier adaptation to changing requirements, technological advancements, or business needs over time.

- **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.

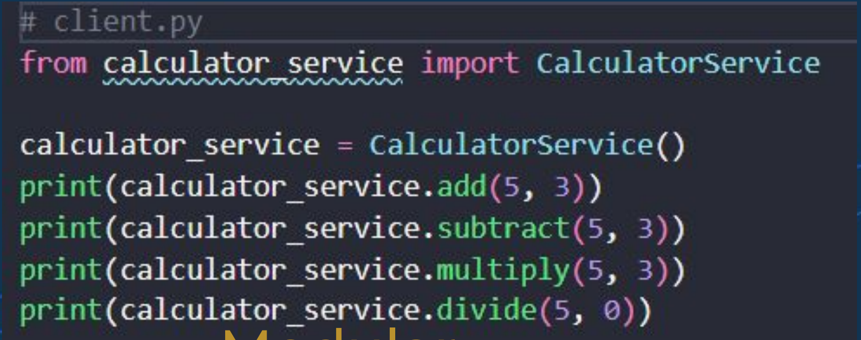
Poll



```python
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, 0))
```



```python
# 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
```



```python
# 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))
```

CoGrammar

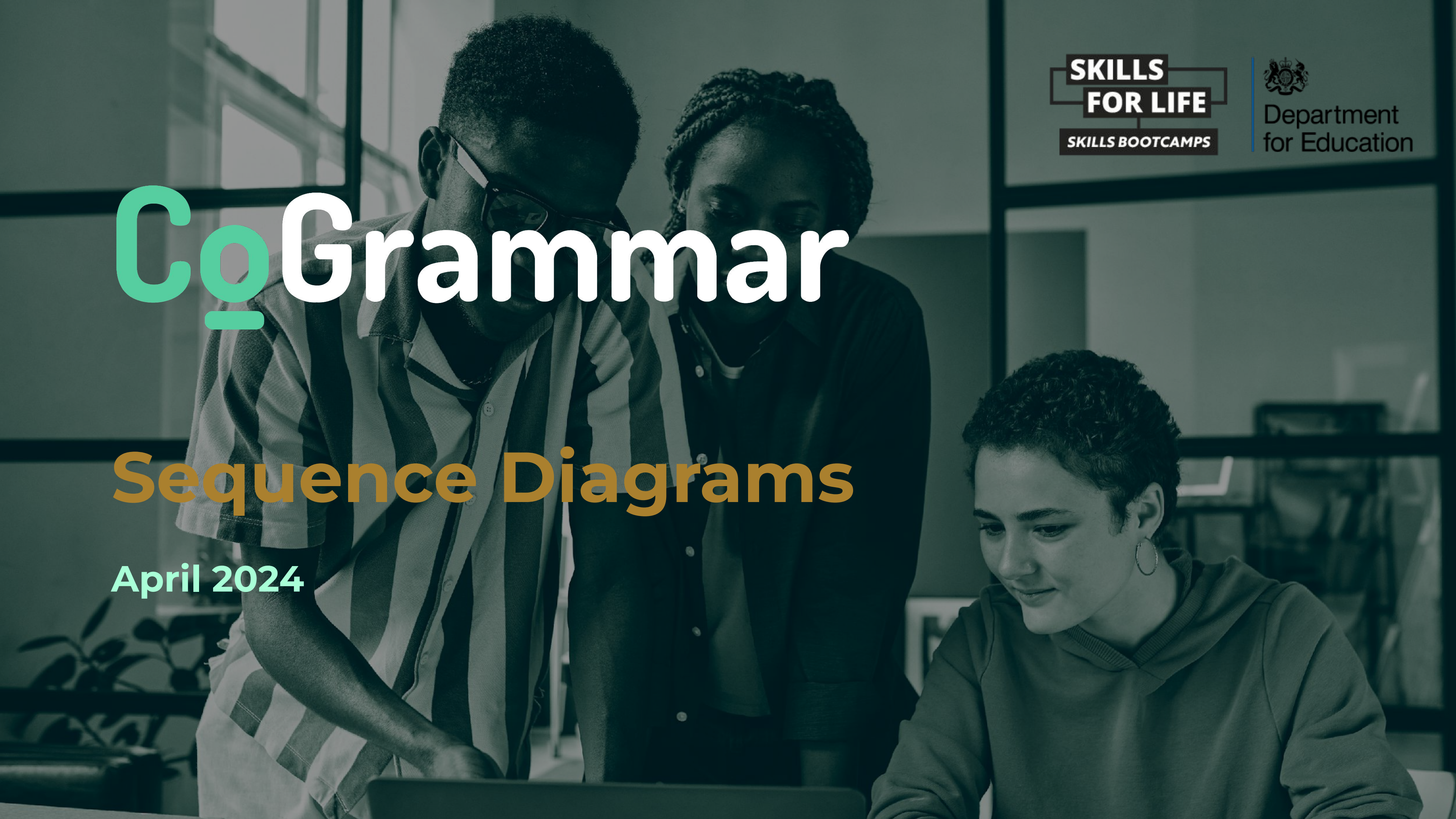
Modular Programming

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!

CoGrammar



SKILLS FOR LIFE

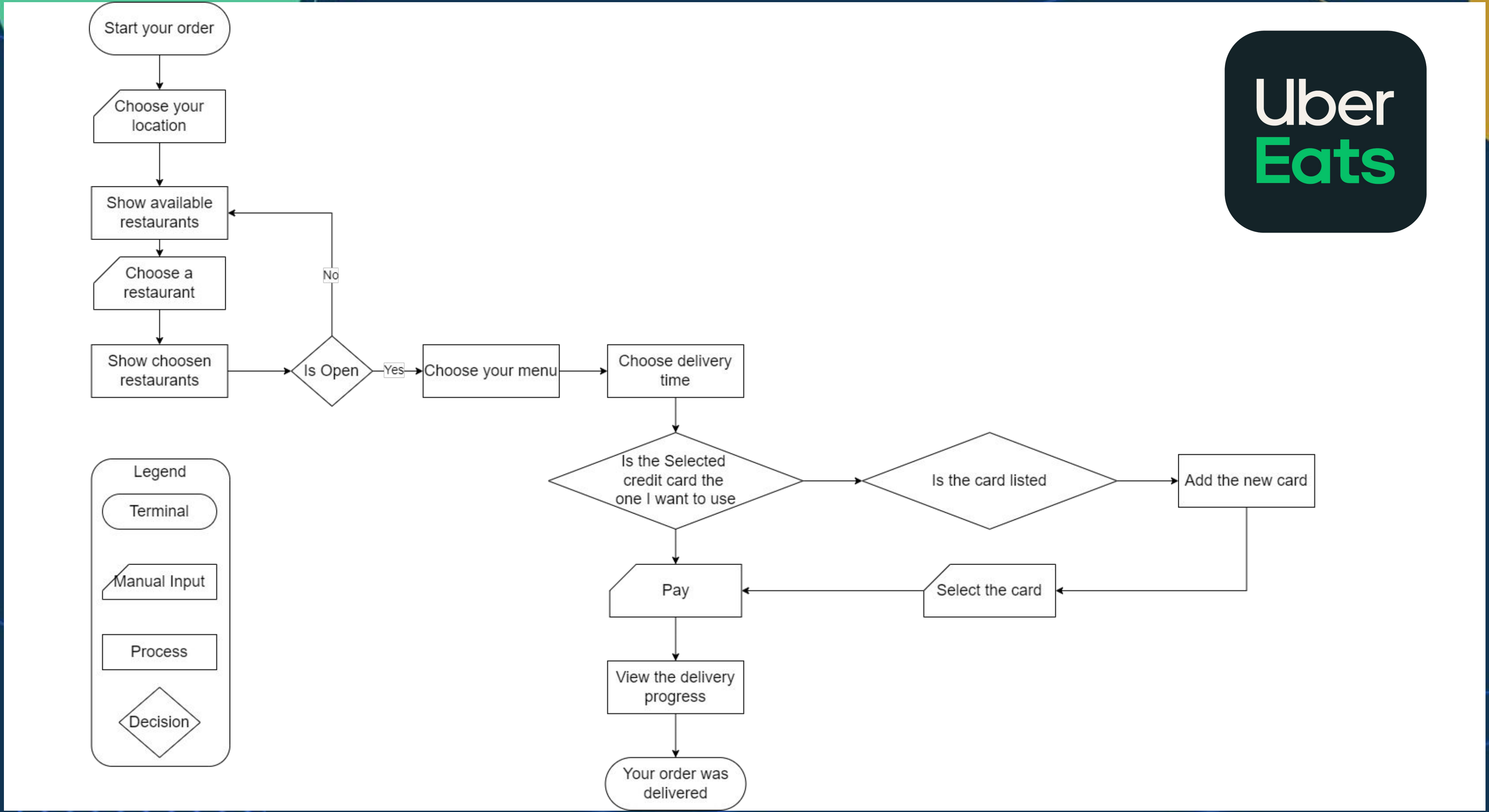
SKILLS BOOTCAMPS

Department for Education

CoGrammar

Sequence Diagrams

April 2024



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.

CoGrammar

Sequence Diagrams Basics

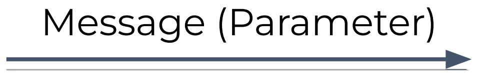
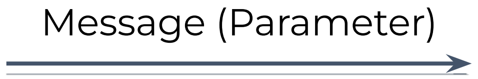
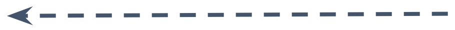
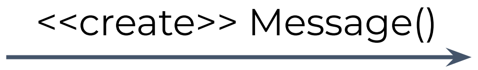
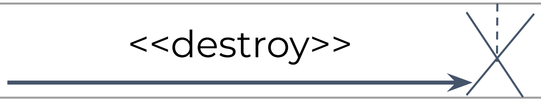
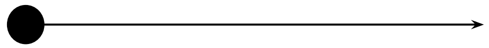
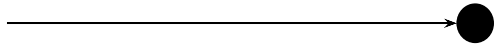
- Shows control flow, the order of interactions
- Time runs vertically, from top to bottom
- Messages run horizontally
- Type of UML diagram

CoGrammar

Poll

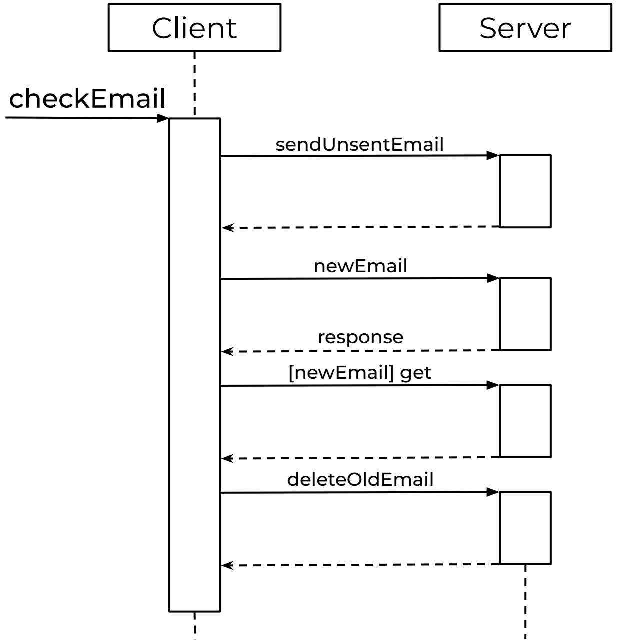
- Given the definition, what do you think are linked to sequence diagrams (1 answer)?
  - Sequence diagram can replace code
  - Sequence diagrams are language-agnostic
  - Representative of the entire software
  - Show relationship between object

Sequence Diagrams Key Components

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

Syntax and semantics

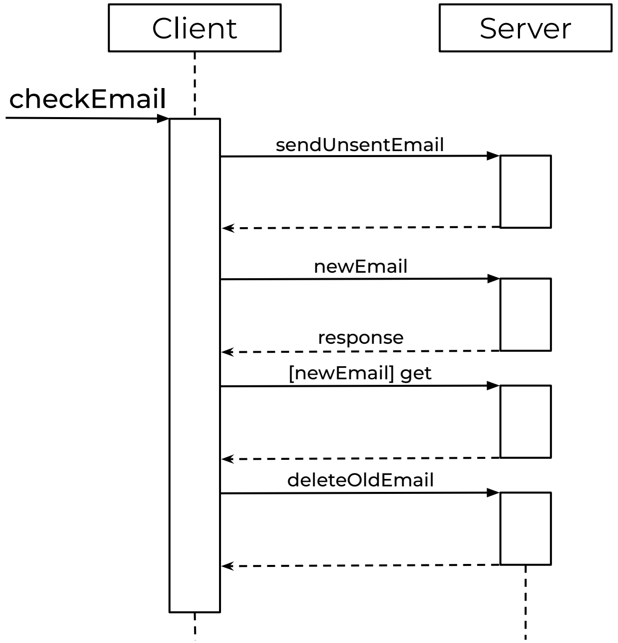
- **Participant:** an object or an entity; the sequence diagram actor
  - sequence diagram starts with an unattached "found message" arrow



CoGrammar

Syntax and semantics

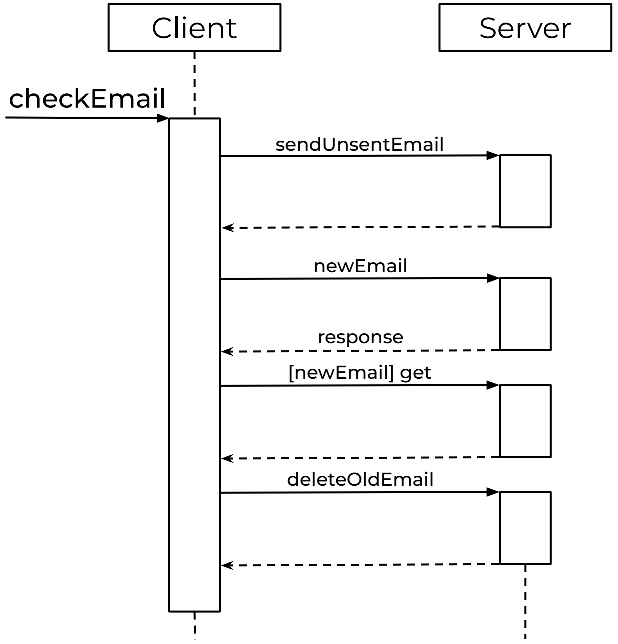
- **Participant:** an object or an entity; the sequence diagram actor
    - sequence diagram starts with an unattached "found message" arrow
- **Message**: communication between objects



CoGrammar

Syntax and semantics

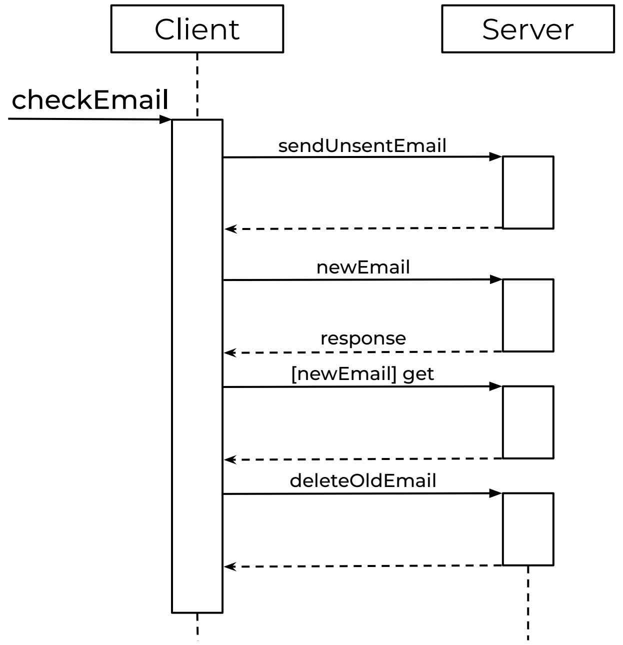
- **Participant:** an object or an entity; the sequence diagram actor
  - sequence diagram starts with an unattached "found message" arrow
- **Message**: communication between objects
- Axes in a sequence diagram:
  - **horizontal**: which participant is acting
  - **vertical**: time (↓ forward in time)



CoGrammar

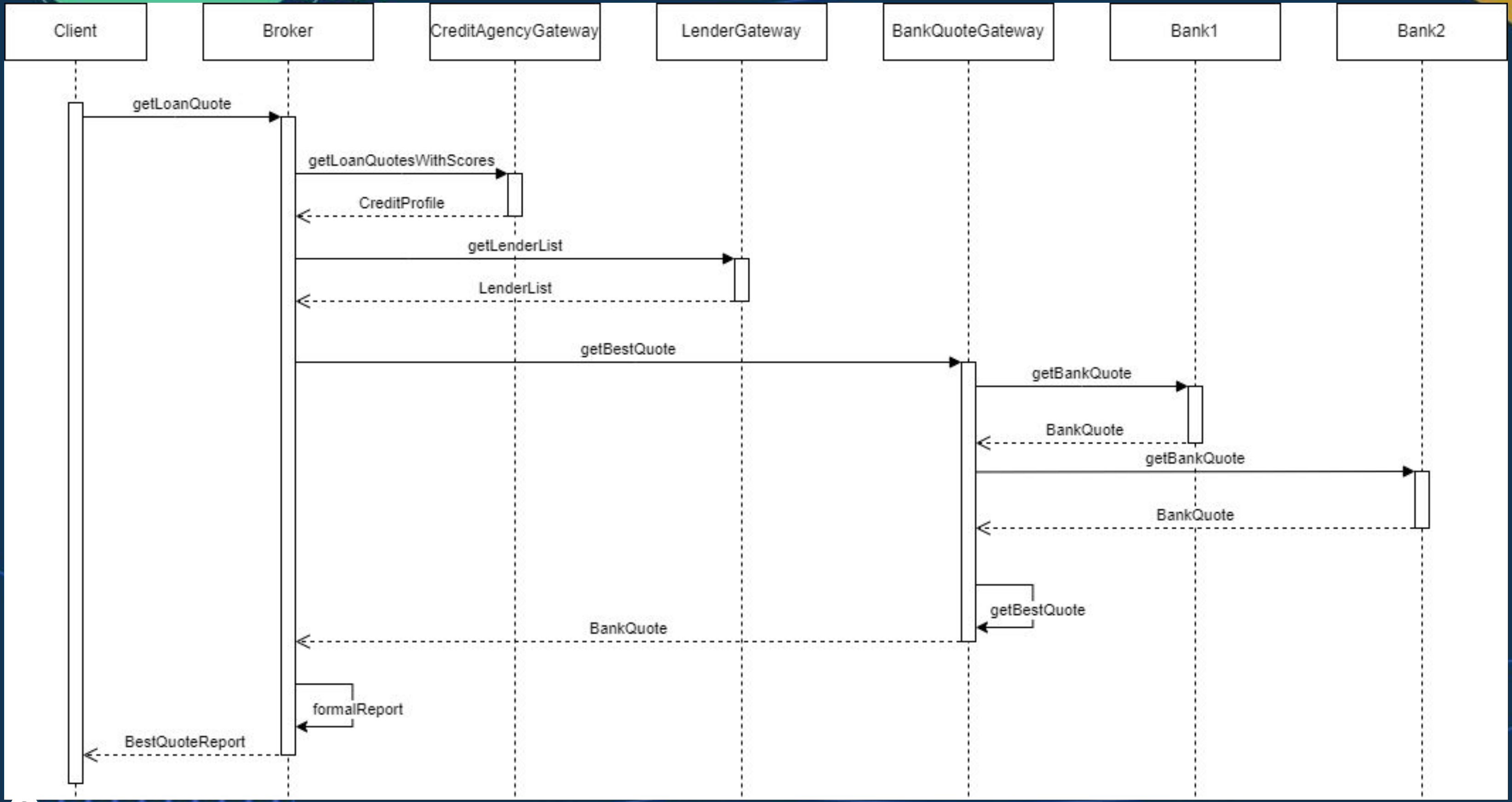
Syntax and semantics

- **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)



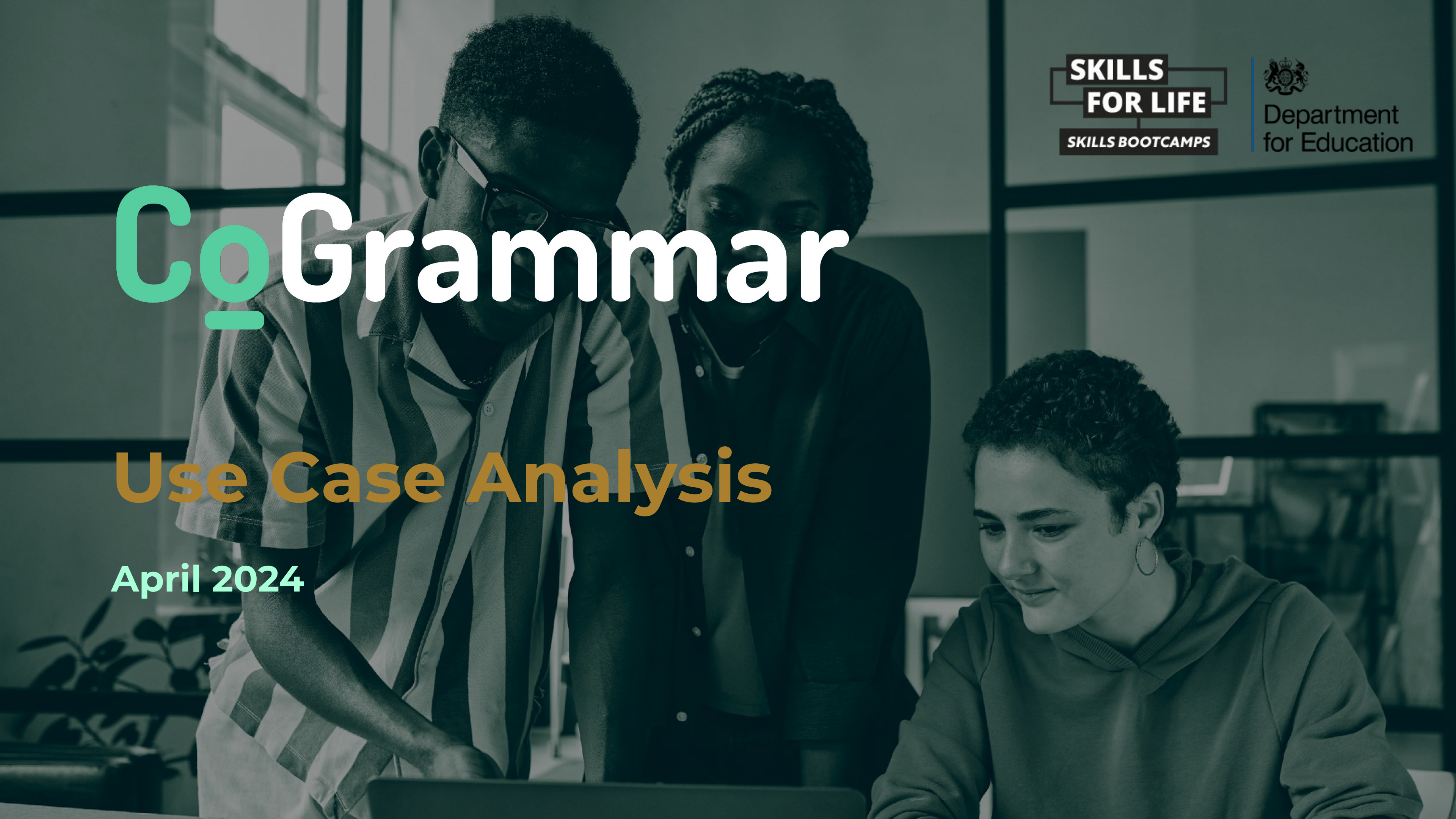
CoGrammar

Use Case



CoGrammar

Sequence Diagrams



SKILLS FOR LIFE

SKILLS BOOTCAMPS

Department for Education

CoGrammar

Use Case Analysis

April 2024

Intuition

However, behind the scenes, there's a complex process involving various stakeholders and systems. Use case diagrams provide a bird's-eye view of this process, capturing the different interactions between actors (such as customers, loan officers, and administrators) and the system itself.

By visualizing the different scenarios and functionalities required to facilitate loan processing, use case diagrams help us understand the core functionalities of the Netflix and how different actors interact with it to achieve their goals efficiently.

CoGrammar

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

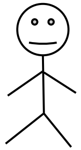
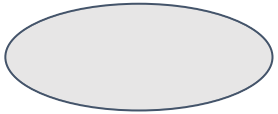
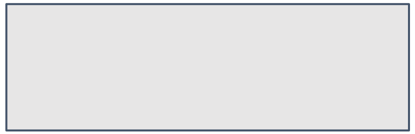
CoGrammar

Poll

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

- Use case diagrams visualise interactions between actors and the loan system.
- Use case diagrams depict the sequence of events in loan processing.
- Use case diagrams represent the internal structure of the loan system.
- Use case diagrams show the implementation details of the loan system.
- Use case diagrams are used primarily for designing graphical user interfaces.

Use Case Diagrams Key Components

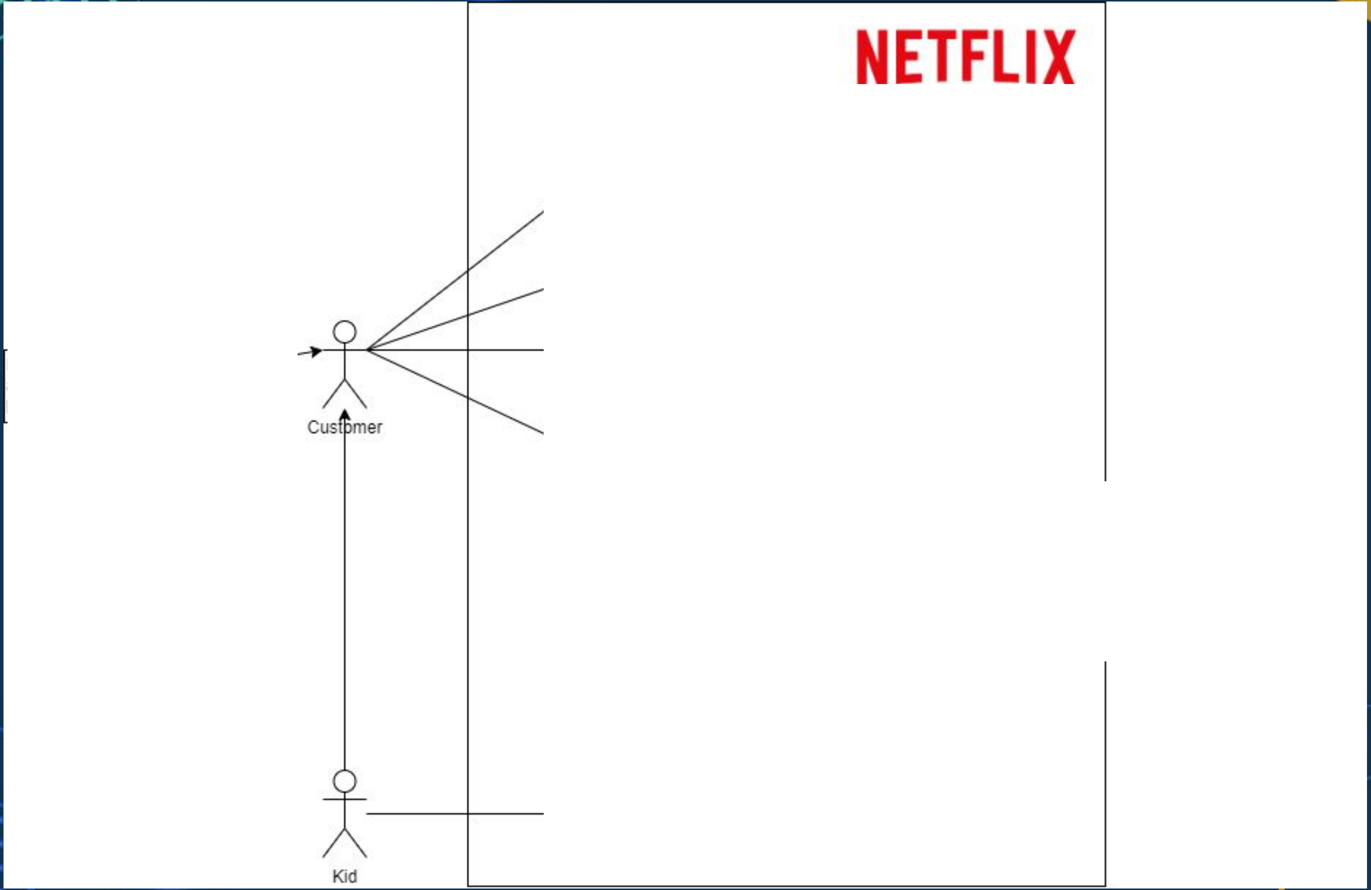
Actor		Human or system interacting with the system. People, Devices, External Systems
Use Cases		Documents the system behaviour from the actor's point of view.
System		Helps identify what is external versus internal, and what the responsibilities of the system are.

CoGrammar

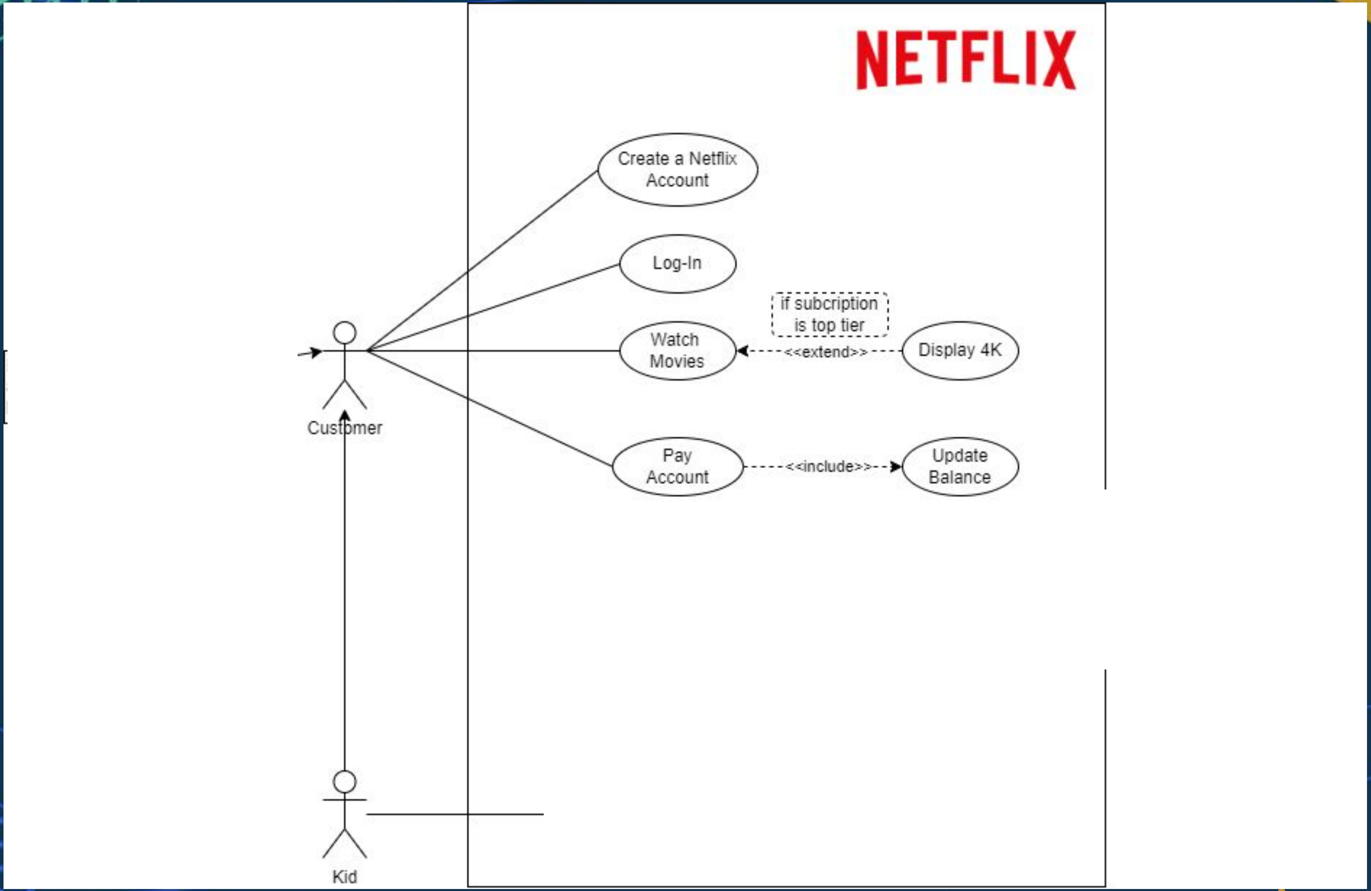
Use Case Diagrams Key Components

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

Syntax and semantics

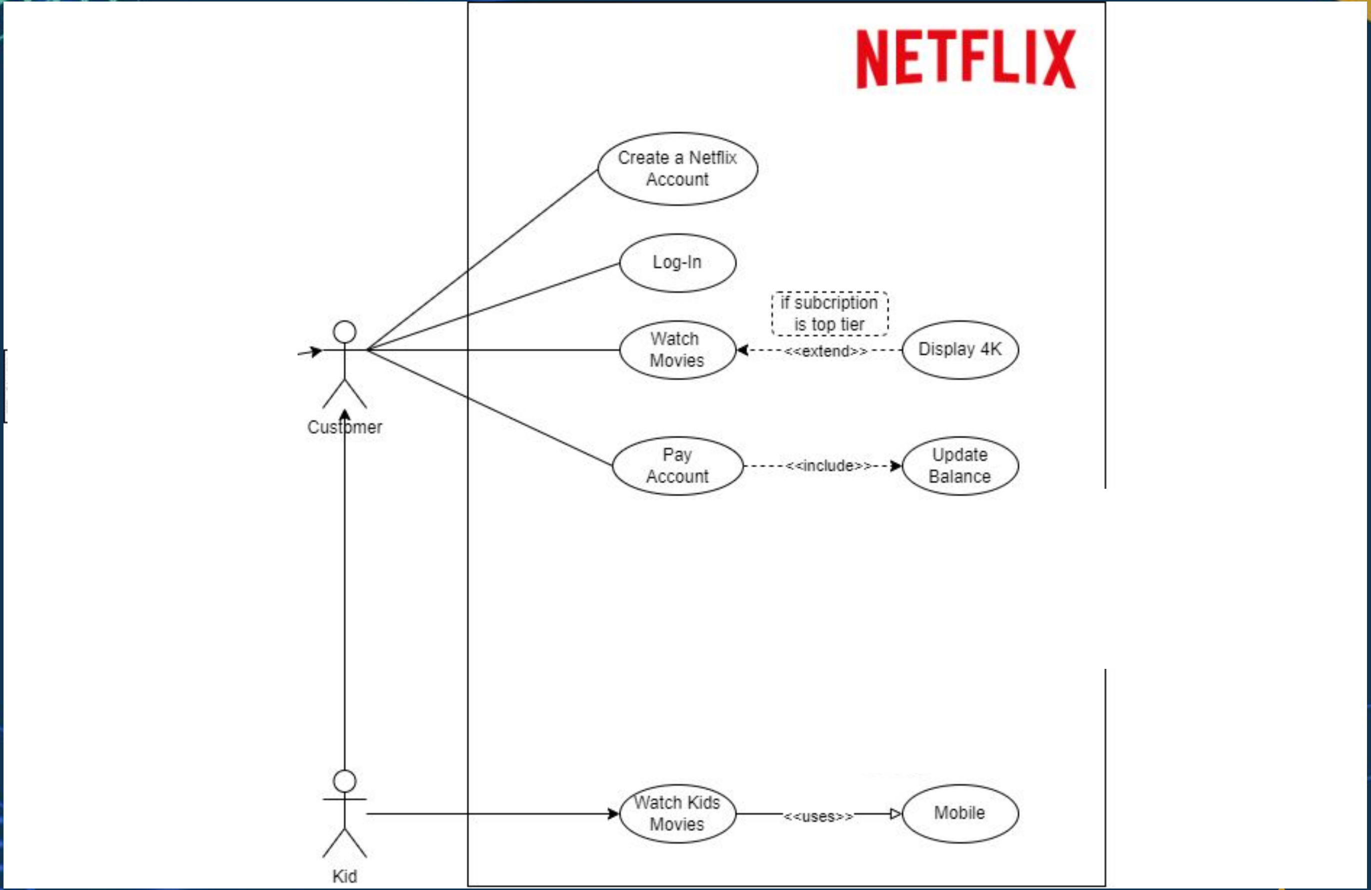


Syntax and semantics



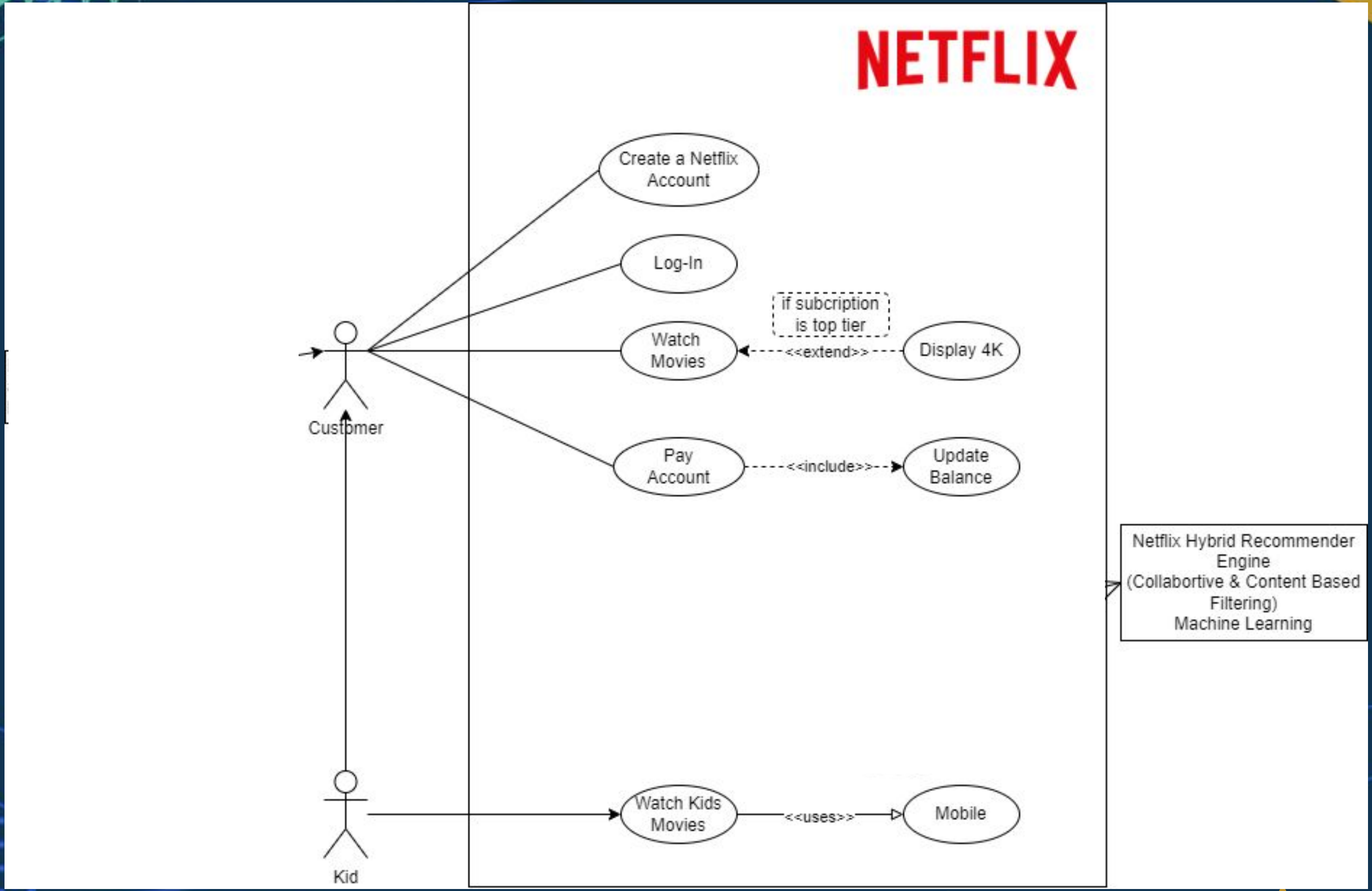
CoGrammar

Syntax and semantics

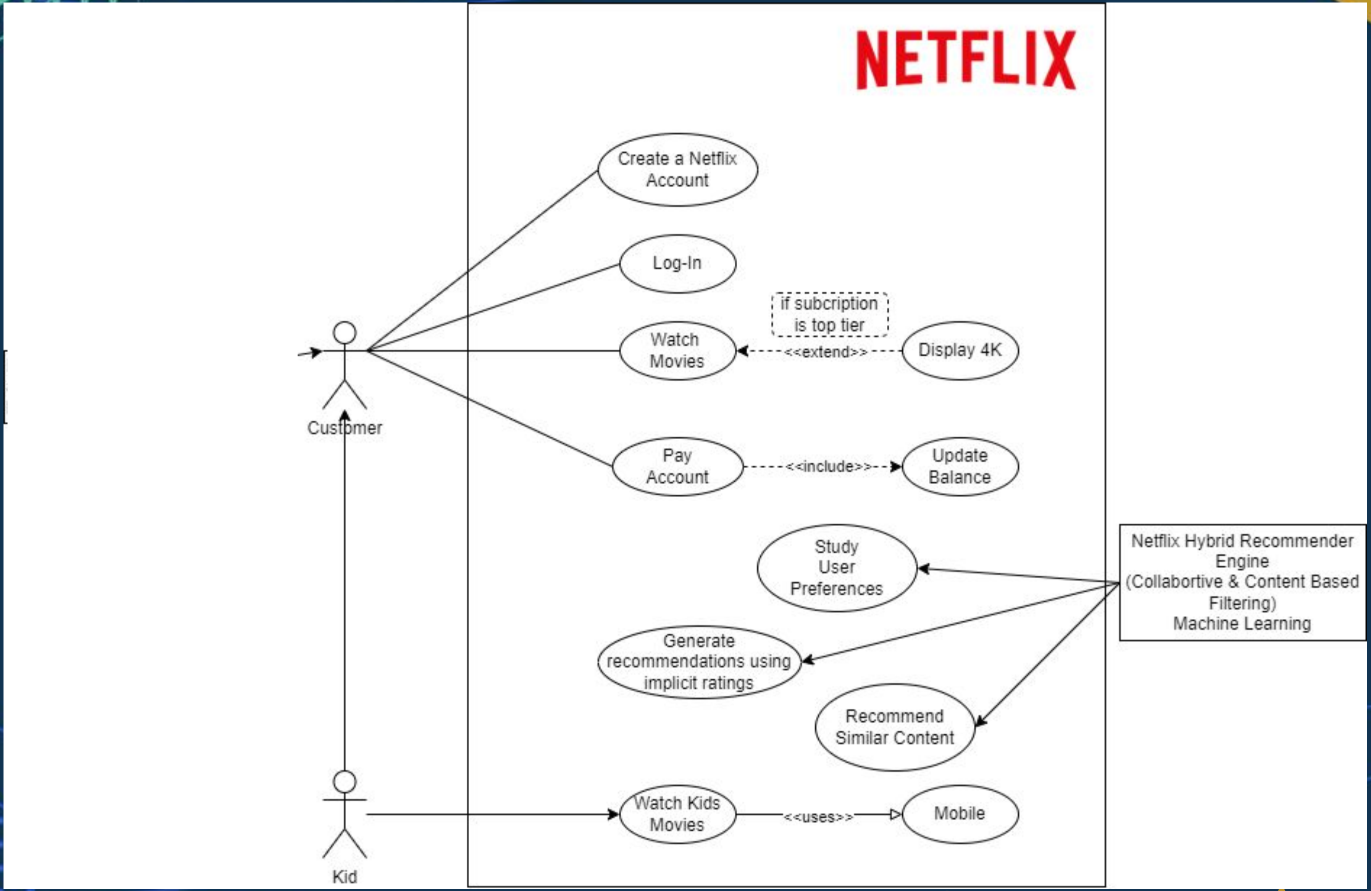


CoGrammar

Syntax and semantics

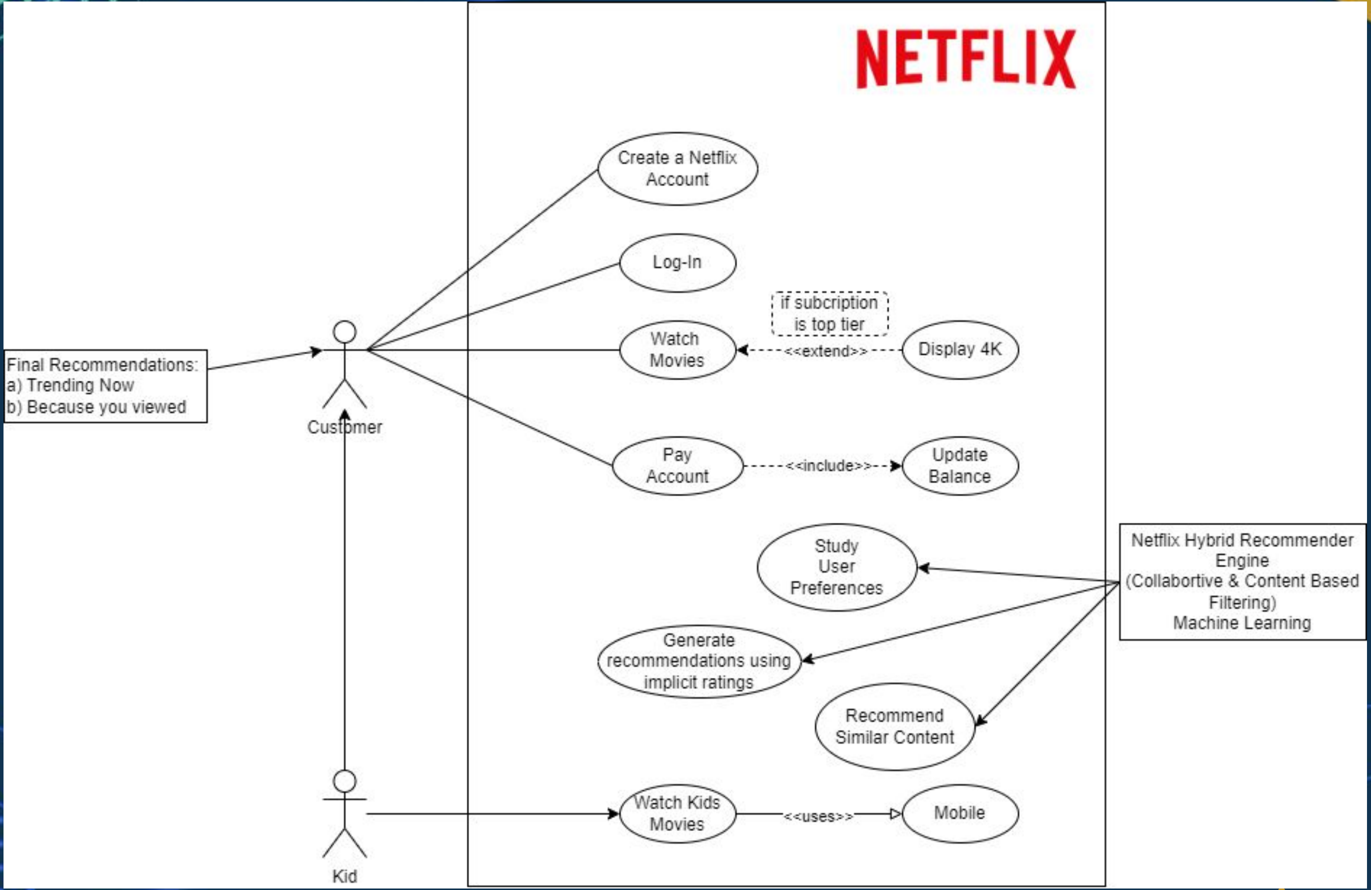


Syntax and semantics


NETFLIX
Create a Netflix Account
Log-In
Watch Movies
if subcription is top tier
<<extend>>
Display 4K
Customer
Pay Account
<<include>>
Update Balance
Study User Preferences
Generate recommendations using implicit ratings
Recommend Similar Content
Netflix Hybrid Recommender Engine (Collabortive & Content Based Filtering) Machine Learning
Watch Kids Movies
<<uses>>
Mobile
Kid

CoGrammar

Syntax and semantics



CoGrammar

Final Polls

- How does modularisation contribute to the scalability of a software system?
    - By reducing coupling between modules
    - By increasing cohesion within modules
    - By enabling independent deployment of modules
    - By improving communication between modules

- In a sequence diagram, what does an arrow represent
    - Data flow
    - Control flow
    - Object instantiation
    - Method invocation

- Which of the following statements accurately describes the purpose of a use case diagram?
    - It shows the interactions between objects within a system.
    - It visualises the flow of messages between system components.
    - It represents the functional requirements of a system from the perspective of its users.
    - It provides a detailed breakdown of system architecture and design.

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.

CoGrammar

Thank you for attending

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar