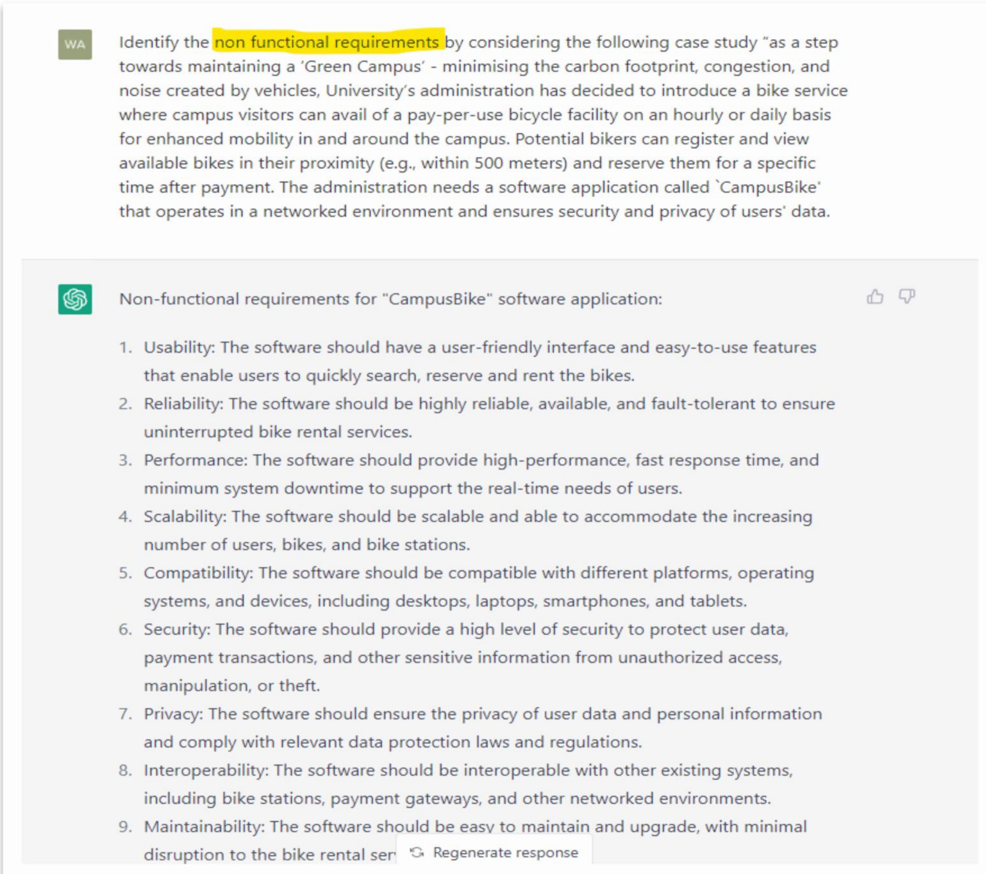
Architectural Synthesis

Non-functional requirements:

Architect: Identify the Non-functional requirements by considering the following case study “as a step towards maintaining a ‘Green Campus’ - minimising the carbon footprint, congestion, and noise created by vehicles, University’s administration has decided to introduce a bike service where campus visitors can avail of a pay-per-use bicycle facility on an hourly or daily basis for enhanced mobility in and around the campus. Potential bikers can register and view available bikes in their proximity (e.g., within 500 meters) and reserve them for a specific time after payment. The administration needs a software application called `CampusBike' that operates in a networked environment and ensures security and privacy of users' data.

ChatGPT: Non-functional requirements for "CampusBike" software application:

- Usability: The software should have a user-friendly interface and easy-to-use features that enable users to quickly search, reserve and rent the bikes.
- Reliability: The software should be highly reliable, available, and fault-tolerant to ensure uninterrupted bike rental services.
- Performance: The software should provide high-performance, fast response time, and minimum system downtime to support the real-time needs of users.
- Scalability: The software should be scalable and able to accommodate the increasing number of users, bikes, and bike stations.
- Compatibility: The software should be compatible with different platforms, operating systems, and devices, including desktops, laptops, smartphones, and tablets.
- Security: The software should provide a high level of security to protect user data, payment transactions, and other sensitive information from unauthorized access, manipulation, or theft.
- Privacy: The software should ensure the privacy of user data and personal information and comply with relevant data protection laws and regulations.
- Interoperability: The software should be interoperable with other existing systems, including bike stations, payment gateways, and other networked environments.
- Maintainability: The software should be easy to maintain and upgrade, with minimal disruption to the bike rental services.
- Accessibility: The software should be accessible to people with disabilities, including visually impaired and hearing-impaired users, to ensure inclusivity and equal access to bike rental services.



WA

Identify the non functional requirements by considering the following case study "as a step towards maintaining a 'Green Campus' - minimising the carbon footprint, congestion, and noise created by vehicles, University's administration has decided to introduce a bike service where campus visitors can avail of a pay-per-use bicycle facility on an hourly or daily basis for enhanced mobility in and around the campus. Potential bikers can register and view available bikes in their proximity (e.g., within 500 meters) and reserve them for a specific time after payment. The administration needs a software application called `CampusBike' that operates in a networked environment and ensures security and privacy of users' data.

Non-functional requirements for "CampusBike" software application:

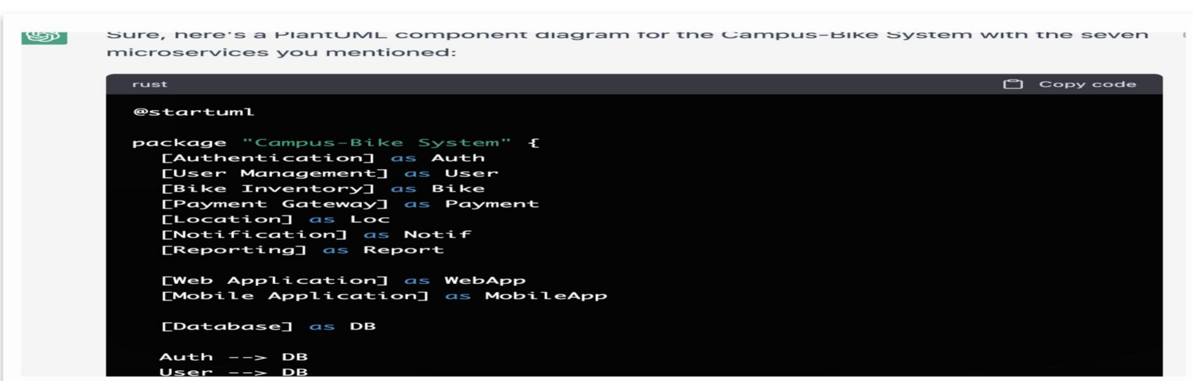
1. Usability: The software should have a user-friendly interface and easy-to-use features that enable users to quickly search, reserve and rent the bikes.
2. Reliability: The software should be highly reliable, available, and fault-tolerant to ensure uninterrupted bike rental services.
3. Performance: The software should provide high-performance, fast response time, and minimum system downtime to support the real-time needs of users.
4. Scalability: The software should be scalable and able to accommodate the increasing number of users, bikes, and bike stations.
5. Compatibility: The software should be compatible with different platforms, operating systems, and devices, including desktops, laptops, smartphones, and tablets.
6. Security: The software should provide a high level of security to protect user data, payment transactions, and other sensitive information from unauthorized access, manipulation, or theft.
7. Privacy: The software should ensure the privacy of user data and personal information and comply with relevant data protection laws and regulations.
8. Interoperability: The software should be interoperable with other existing systems, including bike stations, payment gateways, and other networked environments.
9. Maintainability: The software should be easy to maintain and upgrade, with minimal disruption to the bike rental ser

Regenerate response

Figure 2 Non-Functional Requirements for "CampusBike" software application

Component Diagram:

Architect: Create the Plantuml component diagram for the Campus-Bike System that have following microservices. i) Authentication, ii) User Management, iii) Bike Inventory, iv) Payment Gateway, <v) Location, vi) Notification, and vii) Reporting.



Sure, here's a PlantUML component diagram for the Campus-Bike System with the seven microservices you mentioned:

rust Copy code

```
@startuml

package "Campus-Bike System" {
  [Authentication] as Auth
  [User Management] as User
  [Bike Inventory] as Bike
  [Payment Gateway] as Payment
  [Location] as Loc
  [Notification] as Notif
  [Reporting] as Report

  [Web Application] as WebApp
  [Mobile Application] as MobileApp

  [Database] as DB

  Auth --> DB
  User --> DB
```

Figure 1. PlantUML component diagram script for the CampusBike system

```
@startuml

package "Campus-Bike System" {
  [Authentication] as Auth
  [User Management] as User
  [Bike Inventory] as Bike
  [Payment Gateway] as Payment
  [Location] as Loc
  [Notification] as Notif
  [Reporting] as Report

  [Web Application] as WebApp
  [Mobile Application] as MobileApp

  [Database] as DB

  Auth --> DB
  User --> DB
  Bike --> DB
  Payment --> DB
  Loc --> DB
  Notif --> DB
  Report --> DB

  WebApp --> Auth
  WebApp --> User
  WebApp --> Bike
  WebApp --> Payment
  WebApp --> Loc
  WebApp --> Notif
  WebApp --> Report

  MobileApp --> Auth
  MobileApp --> User
  MobileApp --> Bike
  MobileApp --> Payment
  MobileApp --> Loc
  MobileApp --> Notif
  MobileApp --> Report
}

@enduml
```

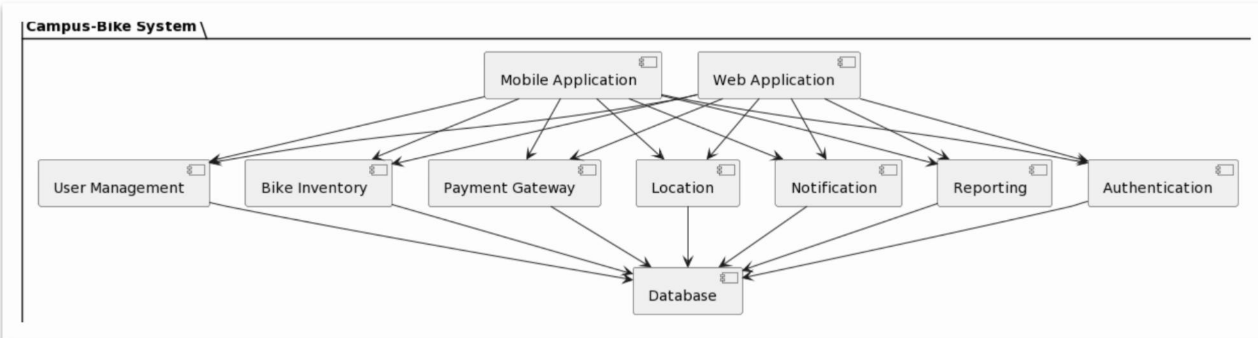


Figure 2 Component diagram generated from the script for the CampusBike system

ChatGPT Explanation: Here's a brief explanation of the diagram:

- The Campus-Bike System is represented as a package, with the seven microservices as components.
- The Web Application and Mobile Application components are also included to show how the different microservices are accessed by the different applications.
- The Database component represents the database used to store the data for the various microservices.
- The microservices are connected to the database and to each other as appropriate for their functionality. For example, the Authentication, User Management, and Bike Inventory microservices all rely on the database for storing and retrieving user and bike data, and the Payment Gateway microservice communicates with an external payment gateway. The Notification microservice uses the database to store notification data, and the Reporting microservice uses the database to retrieve data for generating reports. The Web and Mobile applications both access the various microservices to allow users to manage their accounts, bikes, and payments, as well as receive notifications and view reports.

Class Diagram:

Architect: Create the Plantuml class diagram for the Campus-Bike System (with all types of relationships) that have following microservices. i) Authentication, ii) User Management, iii) Bike Inventory, iv) Payment Gateway, v) Location, vi) Notification, and vii) Reporting.

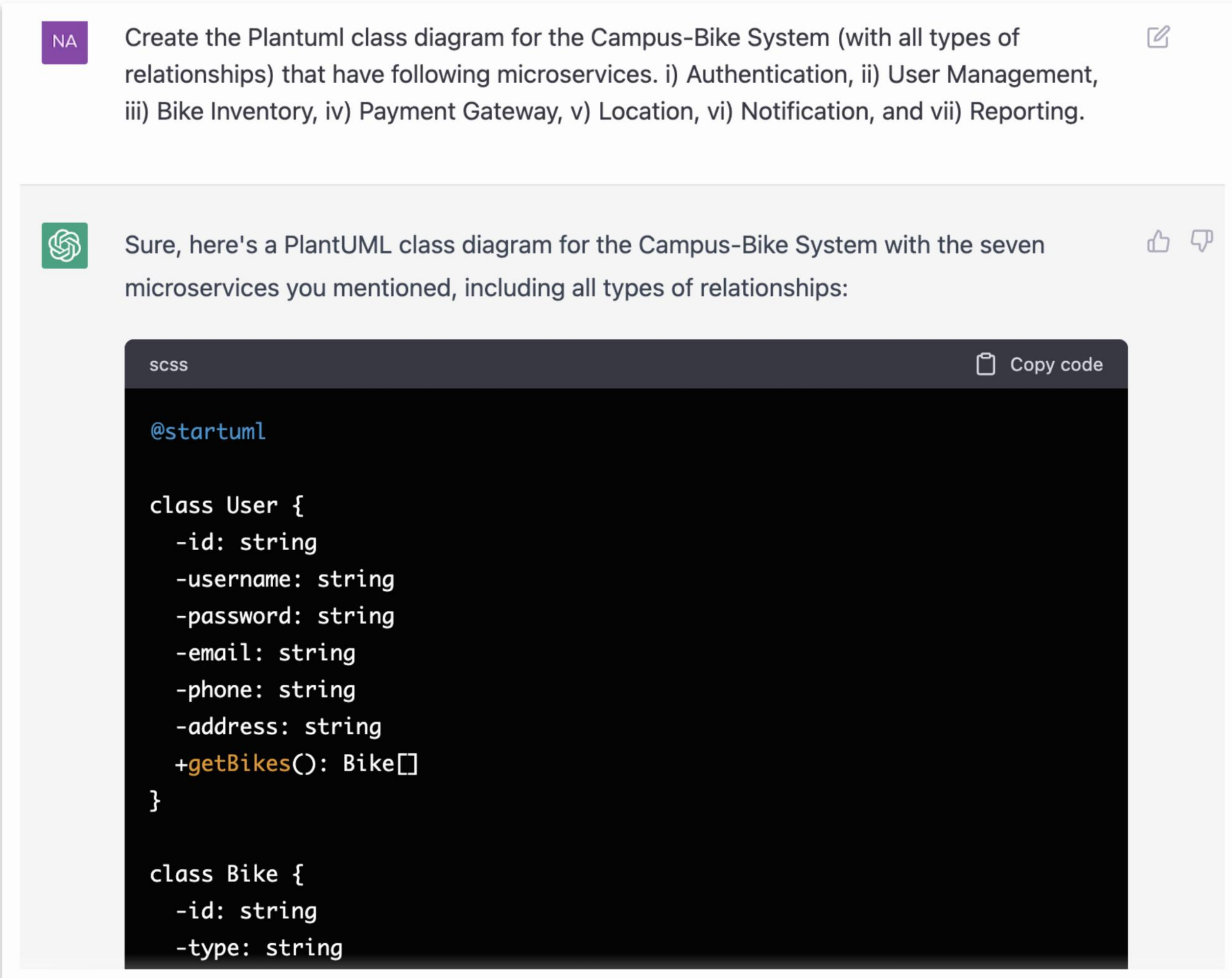


Figure 3. PlantUML class diagram script for the CampusBike system

```
@startuml

class User {
  -id: string
  -username: string
  -password: string
  -email: string
  -phone: string
  -address: string
  +getBikes(): Bike[]
}

class Bike {
  -id: string
  -type: string
  -model: string
```

```
  -status: string
  -location: string
  +getUser(): User
}

class Payment {
  -id: string
  -amount: float
  -status: string
}

class Location {
  -id: string
  -latitude: float
  -longitude: float
}

class Notification {
  -id: string
  -title: string
  -message: string
  -status: string
}

class Report {
  -id: string
  -title: string
  -description: string
  -status: string
}

class Auth {
  +login(username: string, password: string): bool
  +logout(): bool
}

class UserManagement {
  +addUser(user: User): bool
  +deleteUser(user: User): bool
  +updateUser(user: User): bool
}

class BikeInventory {
  +addBike(bike: Bike): bool
  +deleteBike(bike: Bike): bool
  +updateBike(bike: Bike): bool
}

class PaymentGateway {
```

```
  +makePayment(payment: Payment): bool
}

class CampusBikeSystem {
  -auth: Auth
  -userManagement: UserManagement
  -bikeInventory: BikeInventory
  -paymentGateway: PaymentGateway
  -location: Location
  -notification: Notification
  -report: Report
  +login(username: string, password: string): bool
  +logout(): bool
  +addUser(user: User): bool
  +deleteUser(user: User): bool
  +updateUser(user: User): bool
  +addBike(bike: Bike): bool
  +deleteBike(bike: Bike): bool
  +updateBike(bike: Bike): bool
  +makePayment(payment: Payment): bool
}

class WebApplication {
  +displayLoginScreen()
  +displayUserProfile(user: User)
  +displayBikeInventory(bikes: Bike[])
  +displayPaymentGateway()
  +displayLocation()
  +displayNotifications()
  +displayReports()
}

class MobileApplication {
  +displayLoginScreen()
  +displayUserProfile(user: User)
  +displayBikeInventory(bikes: Bike[])
  +displayPaymentGateway()
  +displayLocation()
  +displayNotifications()
  +displayReports()
}

User --> Bike
Bike --> User
Bike --> Location
User --> Payment
Notification --> User
Report --> User
CampusBikeSystem --> Auth
```

CampusBikeSystem --> UserManagement
CampusBikeSystem --> BikeInventory
CampusBikeSystem --> PaymentGateway
CampusBikeSystem --> Location
CampusBikeSystem --> Notification
CampusBikeSystem --> Report
WebApplication --> CampusBikeSystem
MobileApplication --> CampusBikeSystem

@enduml

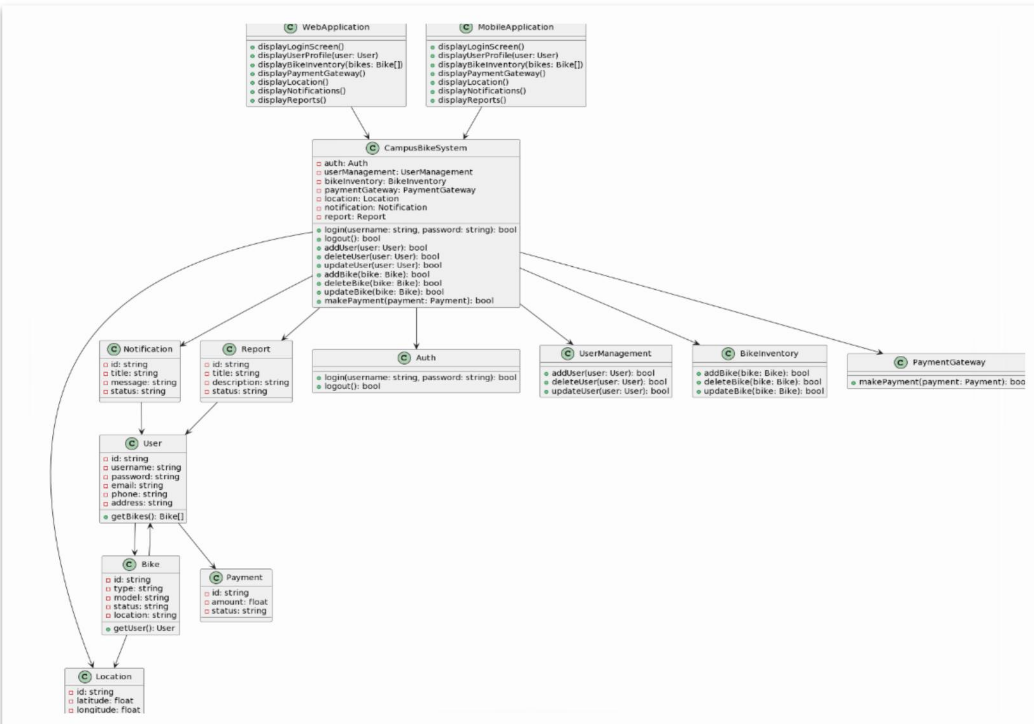


Figure 4. Class diagram generated from the script for the CampusBike system

Implementing Patterns:

1. Singleton Pattern:

The Auth, UserManagement, BikeInventory, PaymentGateway, Location, Notification, and Report classes in CampusBikeSystem can be implemented using the singleton pattern. This ensures that only one instance of these classes can be created, providing a centralized point of access for their functionality.

@startuml

class Auth {
  - static instance: Auth
  - constructor()
  + login(username: string, password: string): bool
  + logout(): bool
  + getInstance(): Auth

```
}

class UserManagement {
  - static instance: UserManagement
  - constructor()
  + addUser(user: User): bool
  + deleteUser(user: User): bool
  + updateUser(user: User): bool
  + getInstance(): UserManagement
}

class BikeInventory {
  - static instance: BikeInventory
  - constructor()
  + addBike(bike: Bike): bool
  + deleteBike(bike: Bike): bool
  + updateBike(bike: Bike): bool
  + getInstance(): BikeInventory
}

class PaymentGateway {
  - static instance: PaymentGateway
  - constructor()
  + makePayment(payment: Payment): bool
  + getInstance(): PaymentGateway
}

class Location {
  - static instance: Location
  - constructor()
  + getInstance(): Location
}

class Notification {
  - static instance: Notification
  - constructor()
  + getInstance(): Notification
}

class Report {
  - static instance: Report
  - constructor()
  + getInstance(): Report
}

class CampusBikeSystem {
  - auth: Auth
  - userManagement: UserManagement
  - bikeInventory: BikeInventory
```

```
  - paymentGateway: PaymentGateway
  - location: Location
  - notification: Notification
  - report: Report
  - constructor()
  + login(username: string, password: string): bool
  + logout(): bool
  + addUser(user: User): bool
  + deleteUser(user: User): bool
  + updateUser(user: User): bool
  + addBike(bike: Bike): bool
  + deleteBike(bike: Bike): bool
  + updateBike(bike: Bike): bool
  + makePayment(payment: Payment): bool
}

Auth --> CampusBikeSystem
UserManagement --> CampusBikeSystem
BikeInventory --> CampusBikeSystem
PaymentGateway --> CampusBikeSystem
Location --> CampusBikeSystem
Notification --> CampusBikeSystem
Report --> CampusBikeSystem

@enduml
```

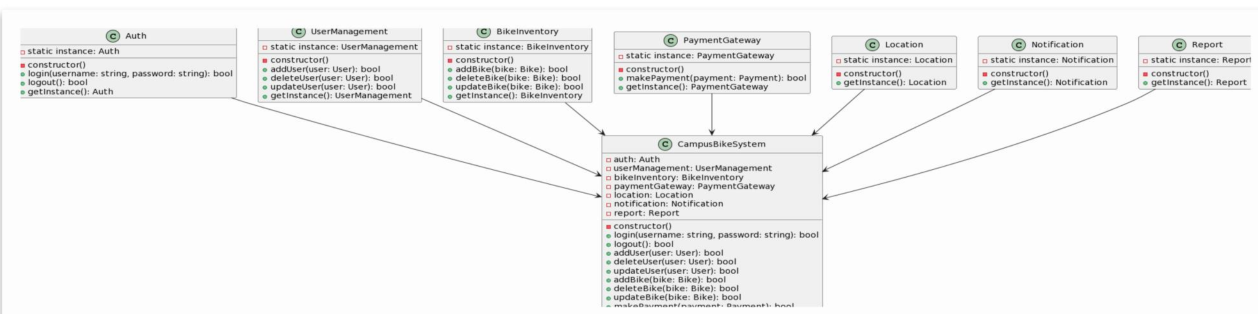


Figure 5. Singleton Pattern Implementation

2. Factory Method Pattern

The WebApplication and MobileApplication classes can implement the factory method pattern to create the appropriate concrete implementation of CampusBikeSystem depending on the type of application that is being used (web or mobile).

```
@startuml

abstract class Application {
```

```
  + displayLoginScreen()
  + displayUserProfile(user: User)
  + displayBikeInventory(bikes: Bike[])
  + displayPaymentGateway()
  + displayLocation()
  + displayNotifications()
  + displayReports()
  + createCampusBikeSystem(): CampusBikeSystem
}

class WebApplication {
  + createCampusBikeSystem(): CampusBikeSystem
}

class MobileApplication {
  + createCampusBikeSystem(): CampusBikeSystem
}

Application <|-- WebApplication
Application <|-- MobileApplication
WebApplication --> CampusBikeSystem
MobileApplication --> CampusBikeSystem

@enduml
```

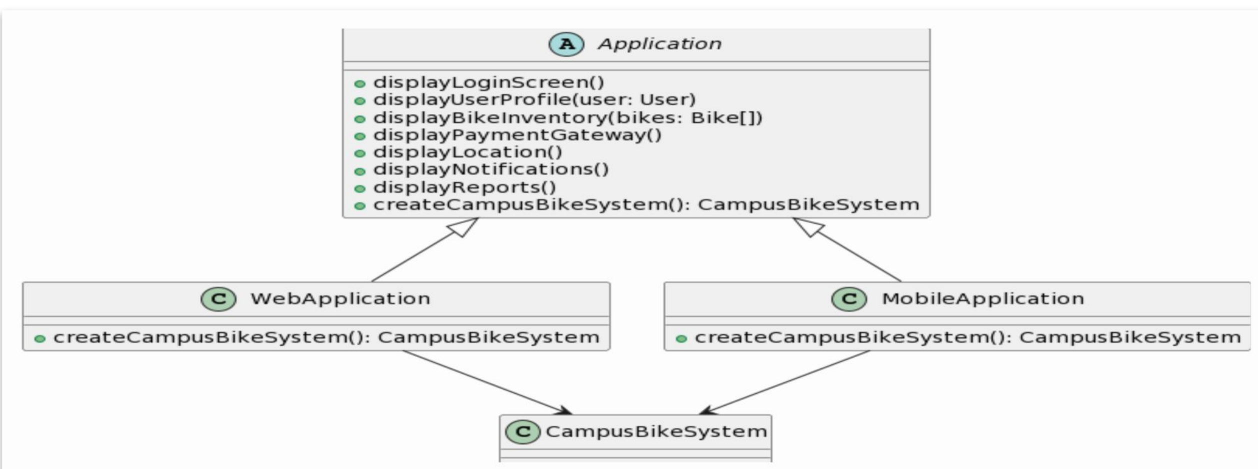


Figure 6 Factory Method Pattern Implementation

3. Observer Pattern

Sure, here is an example of how to apply the observer pattern to the given PlantUML class diagram for the Notification and User classes:

In this implementation, the User class is an observer of the Notification class, and it implements the Observer interface. The Notification class has methods to attach, detach, and notify observers, and it maintains a list of observers. When a new message or alert is created, the Notification class notifies its observers, which includes the User class. The User class is then able to update its internal state accordingly and display the new message or alert to the user.

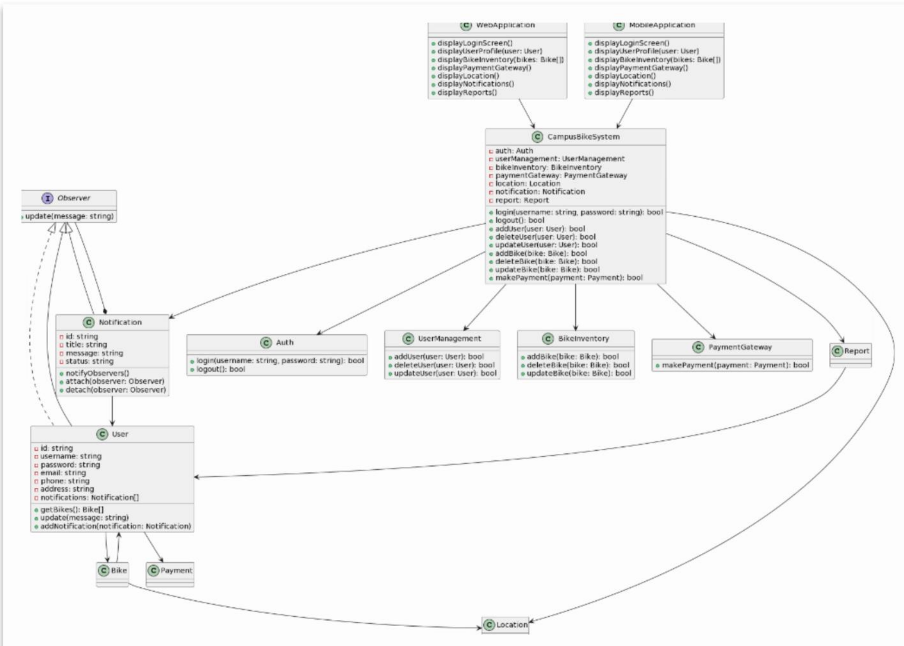


Figure 7 Observer Pattern Implementation

4. Strategy Pattern

The Strategy pattern allows the behavior of an object to be selected at runtime, based on the specific algorithm needed.
In the given class diagram, the Strategy pattern could be applied to the PaymentGateway class. The payment gateway may have different payment strategies that may be implemented for different payment methods. Instead of implementing each payment method in the payment gateway class, a payment strategy interface could be created and implemented by specific payment method classes. The payment gateway can then use the payment strategy interface to execute the specific payment method.
Here's an updated class diagram that implements the Strategy pattern:

```
@startuml

interface PaymentStrategy {
  + makePayment(payment: Payment): bool
}

class CreditCardStrategy {
  + makePayment(payment: Payment): bool
```

```
}

class DebitCardStrategy {
  + makePayment(payment: Payment): bool
}

class Payment {
  - id: string
  - amount: float
  - status: string
}

class PaymentContext {
  - strategy: PaymentStrategy
  + setPaymentStrategy(strategy: PaymentStrategy)
  + makePayment(payment: Payment): bool
}

class CampusBikeSystem {
  - auth: Auth
  - userManagement: UserManagement
  - bikeInventory: BikeInventory
  - paymentGateway: PaymentContext
  - location: Location
  - notification: Notification
  - report: Report
  + login(username: string, password: string): bool
  + logout(): bool
  + addUser(user: User): bool
  + deleteUser(user: User): bool
  + updateUser(user: User): bool
  + addBike(bike: Bike): bool
  + deleteBike(bike: Bike): bool
  + updateBike(bike: Bike): bool
  + makePayment(payment: Payment): bool
}

PaymentContext --> PaymentStrategy
PaymentContext *-- Payment
PaymentContext ..> PaymentGateway
PaymentGateway ..> PaymentStrategy
PaymentStrategy <|.. CreditCardStrategy
PaymentStrategy <|.. DebitCardStrategy

@enduml
```

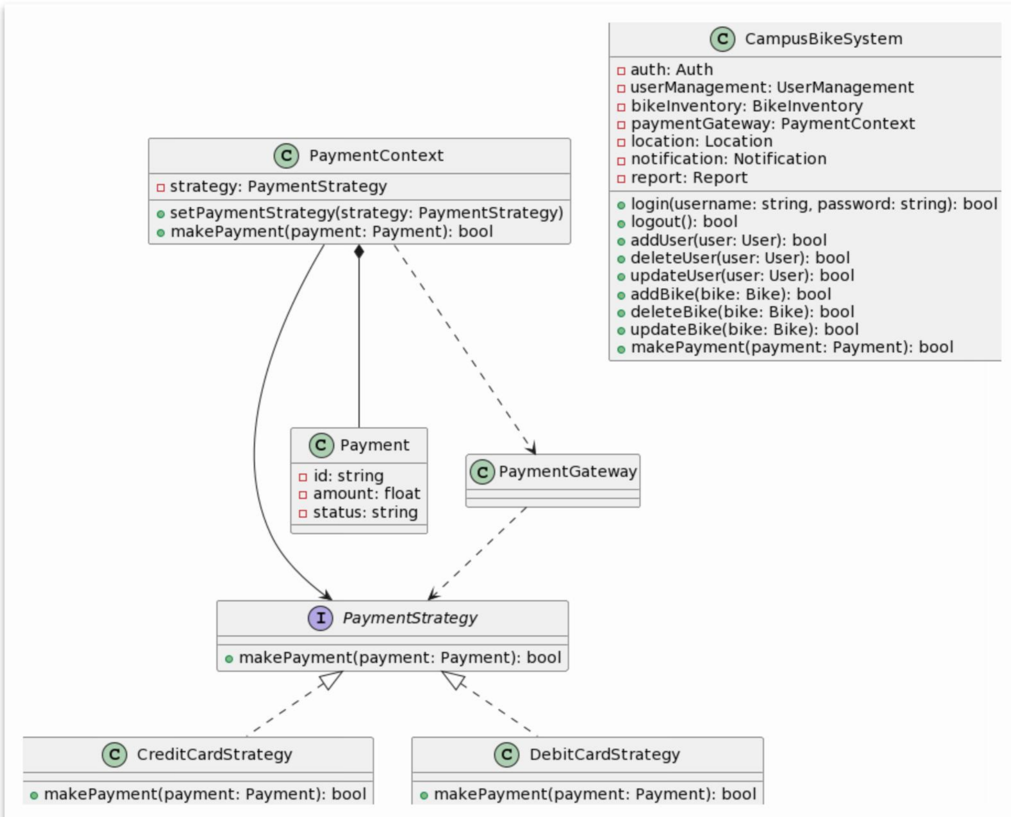


Figure 8. Strategy Pattern implementation

5. Decorator Pattern

In the given class diagram, the Decorator pattern can be applied to the Bike class to add additional features to a bike.
The UML class diagram for the Decorator pattern implementation is:

```
@startuml
class Bike {
  - id: string
  - type: string
  - model: string
  - status: string
  - location: string
  + getUser(): User
  + getPrice(): float
}

class BikeFeatureDecorator {
  - bike: Bike
  + getPrice(): float
}
```

```
class BikeAccessoryDecorator {
  - bike: Bike
  + getPrice(): float
}

class BikeInsuranceDecorator {
  - bike: Bike
  + getPrice(): float
}

BikeFeatureDecorator --> Bike
BikeAccessoryDecorator --> Bike
BikeInsuranceDecorator --> Bike

@enduml
```

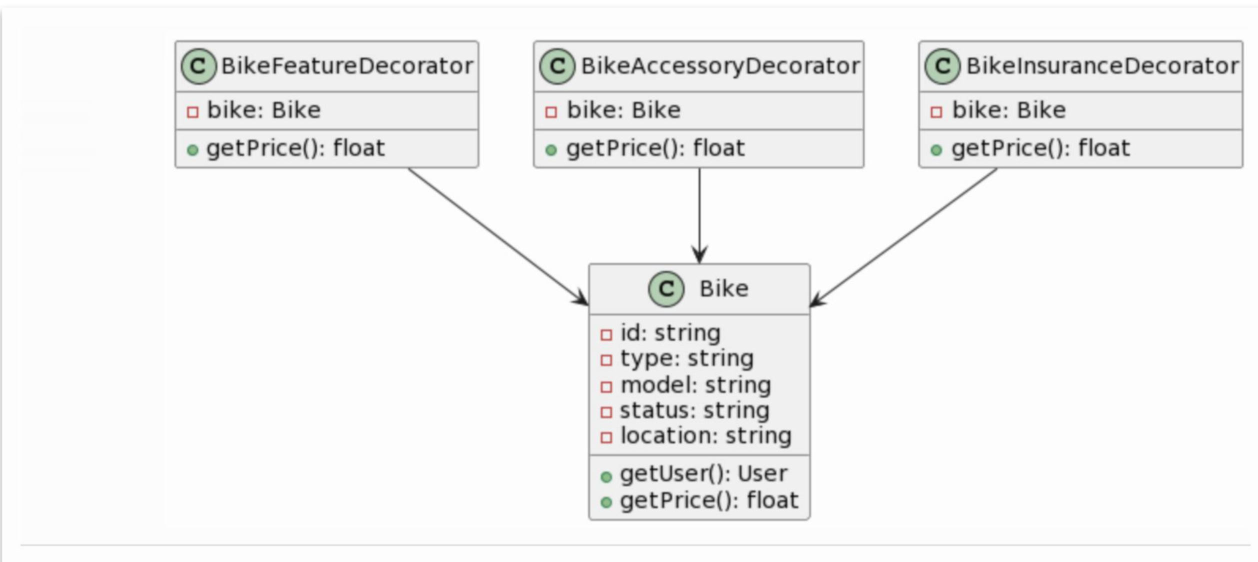


Figure 9 Decorator Pattern Implementation

Here, BikeFeatureDecorator, BikeAccessoryDecorator, and BikeInsuranceDecorator are decorators that can be used to add additional features to the Bike class. All decorators extend the Bike class and have a bike attribute, which stores the decorated bike. They also override the getPrice() method of the Bike class to return the price of the decorated bike.
The Bike class now has a getPrice() method that returns the price of the bike. This method can be overridden by the decorator classes to add additional features to the bike.
For example, a BikeFeatureDecorator can be used to add a feature like a lock to the bike. The getPrice() method of the BikeFeatureDecorator can add the cost of the lock to the price of the bike.
Similarly, a BikeAccessoryDecorator can be used to add accessories like a basket or a bell to the bike. The BikeInsuranceDecorator can be used to add an insurance cost to the bike.
By using the decorator pattern, it is possible to add new features to the bike dynamically, without modifying the original Bike class.