Software Design and Development

Design Document

November 3rd, 2023

Valet Buddy

Airport Valet Car Locator Mobile App

VALET BUDDY

Project Team

Our team is Team 1 comprised of the following students:

- **Jordan Treutel**
- **Richard Hoehn**
- **Ian Hurd**
- **Patrick Burnett**

Introduction

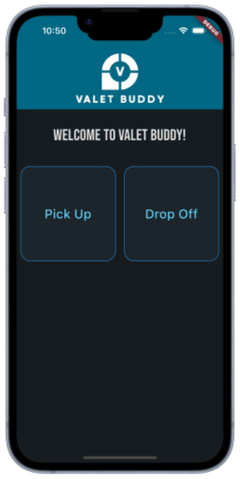
This document serves as the basis for the UML Class diagrams that are based on the Object-Oriented Design (OOD) principles. OOD plays a crucial role in building modular and maintainable software systems, which emphasizes the importance of structuring code in a way that promotes reusability, scalability, and robustness. By encapsulating data and behavior into objects and defining clear interfaces through classes in the Valet Buddy App, we can enable developers to create components that can be easily understood, modified, and extended, which reduces the complexity of software development.

The use of inheritance and polymorphism in object-oriented design further enhances modularity, as it allows for the creation of flexible of the Valet Buddy App where new functionalities can be added with low impact on existing code. This not only speeds up the development process but also ensures that the software can adapt to changing requirements over time, ensuring its long-term viability.

Moreover, the emphasis on design patterns in OOD provides a shared vocabulary and proven solutions to common problems, helping in communication among developers and promoting best practices. This results in more consistent and error-free code, making the system easier to maintain and debug.

In summary, object-oriented design is very important in building software systems that are modular, maintainable, and capable of standing the test of time, providing a solid foundation for sustainable software development.

Intro to App / Project



The project is a mobile app for iOS and Android that allows a Traveler (car owner) to drop off their car with a Valet (driver) that parks the car and sets a GPS location for the parked car based on their phone's current GPS location. In addition to the car's GPS location the associated license plate number, name of the car owner, and picture of the car are also captured during the "Drop-Off" phase.

Another user (Pick-Up-User) can then retrieve the location of the car by entering the license plate number. The storing and retrieval of the car is facilitated by a server that accesses a database for getting and setting GPS coordinates, the license plate and image for later retrieval.

With this app (Valet Buddy), GPS can be used to "Set" and "Get" the car's location and displayed on a user's mobile phone.

The business case of this app is simple... Using technology (specifically GPS) the Airport Car parking owner will not have to "update" and paint numbers on the parking spaces anymore, saving time, and money. Instead, they can rely on the "Valet Buddy" app and backend server to keep track of all the cars that are parked on their lot and quickly and efficiently retrieve them when a car owner returns from the travels at the airport.

Intended Audience

The intended audience of a document refers to the specific group or individuals for whom the document is created or intended.

For this Design Document, the intended audience would be the development company responsible for creating Valet Buddy. This includes:

- **Software Developers:**
  - Who initially implement the system
  - Who maintain the over Valet Buddy application
  - Of neighboring systems who need to know about external interfaces and their technical details
- **Software Architects:**
  - Who need to prepare, shepherd, or implement architectural decisions
  - Who do the design reviews for SOX3 and Security Compliance
- **Quality Assurance Testers:**
  - Who need to perform unit and integration testing within the system
  - Re-Check after features are extended on the Valet Buddy

Object-Oriented Principles

Encapsulation

Data Encapsulation describes the idea of bundling certain methods and attributes together. This is done by creating classes. Encapsulation allows for the hiding of data within an object's internal state. Hiding data and methods within an object restricts how a user can interact with the object.

Inheritance

One of the core principles of object-oriented design, inheritance allows for the creation of a class that is based on an existing class. This is done using parent and child classes, with the child inheriting the parent's properties and behaviors. The benefits of introducing inheritance to a system include code reusability, hierarchical class structure, ability to represent the "is-a" relationship between classes, and polymorphism.

Polymorphism

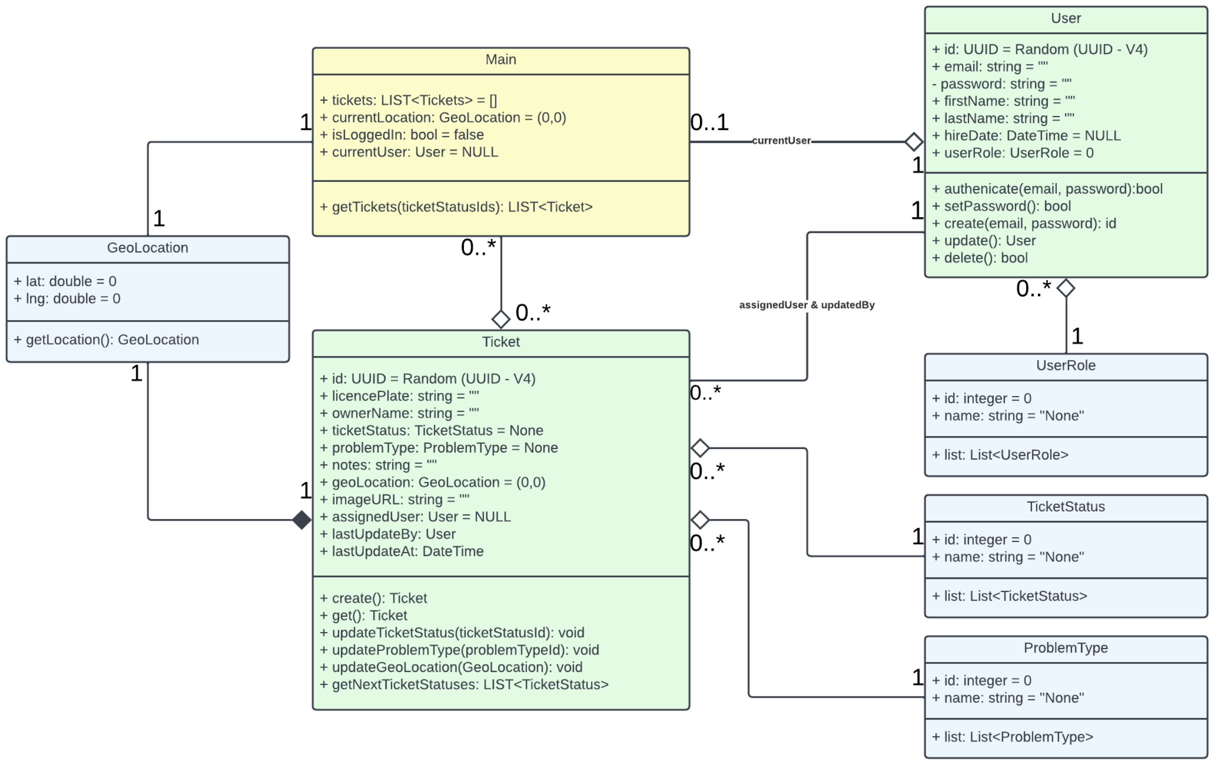
Closely linked with inheritance, polymorphism allows for objects of different classes to be treated as objects of the same base/parent class. This means that regardless of class type, different types of the same base class can be treated in the same consistent and generic way. The components and benefits of polymorphism include method overriding, interfaces/abstract classes, and in general – code flexibility and reusability.

Abstraction

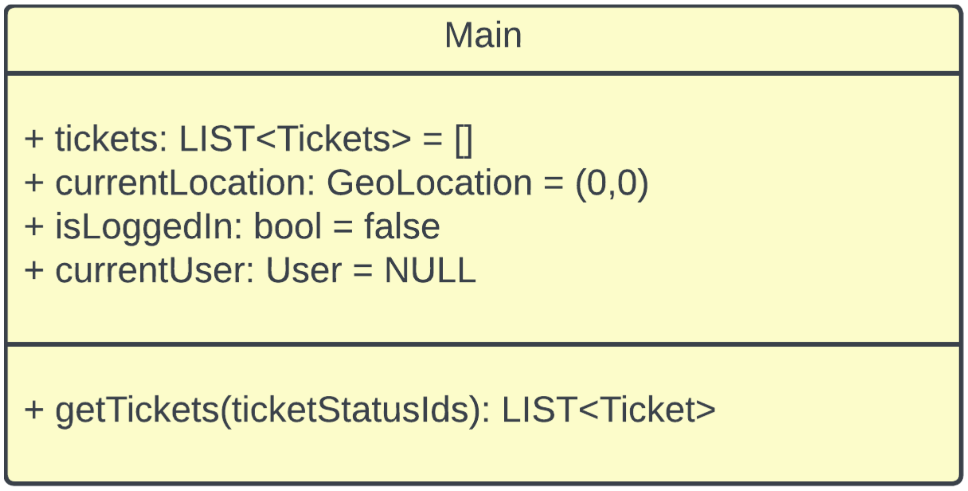
Abstraction is the process of simplifying a real-life object or idea into a model that contains only relevant and essential details, while removing the unnecessary ones. This allows for a high-level representation of each component in a system, focusing on how it behaves, rather than details on how it is implemented. Abstraction allows for hiding the complexity of objects and provides a simplified interface for interacting with them. Hiding the complexity makes the code more readable, writable, and easier to maintain over time.

Class Design

Class Diagram:


User
+ id: UUID = Random (UUID - V4)
+ email: string = ""
- password: string = ""
+ firstName: string = ""
+ lastName: string = ""
+ hireDate: DateTime = NULL
+ userRole: UserRole = 0
+ authenicate(email, password):bool
+ setPassword(): bool
+ create(email, password): id
+ update(): User
+ delete(): bool
Main
+ tickets: LIST<Tickets> = []
+ currentLocation: GeoLocation = (0,0)
+ isLoggedIn: bool = false
+ currentUser: User = NULL
+ getTickets(ticketStatusIds): LIST<Ticket>
GeoLocation
+ lat: double = 0
+ lng: double = 0
+ getLocation(): GeoLocation
Ticket
+ id: UUID = Random (UUID - V4)
+ licencePlate: string = ""
+ ownerName: string = ""
+ ticketStatus: TicketStatus = None
+ problemType: ProblemType = None
+ notes: string = ""
+ geoLocation: GeoLocation = (0,0)
+ imageURL: string = ""
+ assignedUser: User = NULL
+ lastUpdateBy: User
+ lastUpdateAt: DateTime
+ create(): Ticket
+ get(): Ticket
+ updateTicketStatus(ticketStatusId): void
+ updateProblemType(problemTypeId): void
+ updateGeoLocation(GeoLocation): void
+ getNextTicketStatuses: LIST<TicketStatus>
UserRole
+ id: integer = 0
+ name: string = "None"
+ list: List<UserRole>
TicketStatus
+ id: integer = 0
+ name: string = "None"
+ list: List<TicketStatus>
ProblemType
+ id: integer = 0
+ name: string = "None"
+ list: List<ProblemType>
currentUser
assignedUser & updatedBy
1
1
0..1
1
1
0..*
0..*
0..*
1
1
1
0..*
0..*
1
0..*
1
1

Main:



```
Main

+ tickets: LIST<Tickets> = []
+ currentLocation: GeoLocation = (0,0)
+ isLoggedIn: bool = false
+ currentUser: User = NULL

+ getTickets(ticketStatusIds): LIST<Ticket>
```

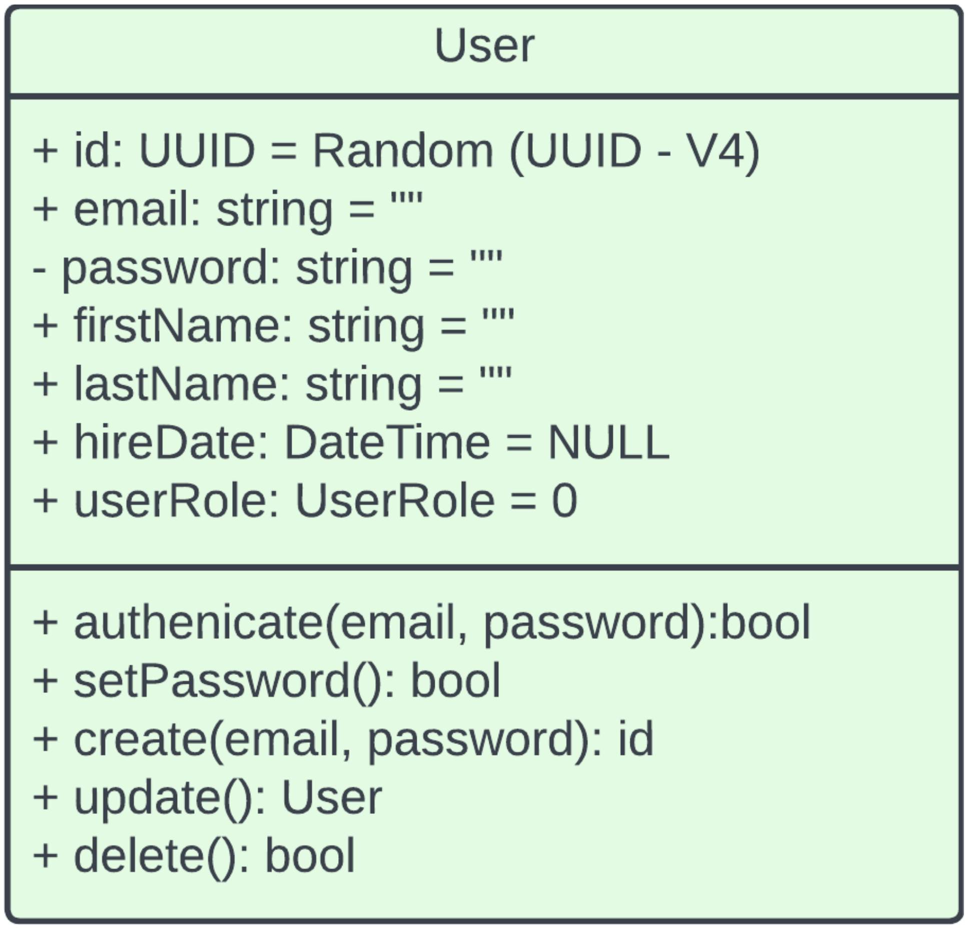
Responsibility

The main instance is a type of singleton – meaning there is only one instance of "Main" in our system. The is the key class instance that holds the list of tickets that the user has access to.

Relationships:

- Ticket
  - Can access ticket table of many tables any ticket and can be used by user to assign a ticket.
- User
  - Authenticates user, and can be used to assign tickets t a user.
- GeoLocation
  - Can retrieve geolocation data, and send it to ticket

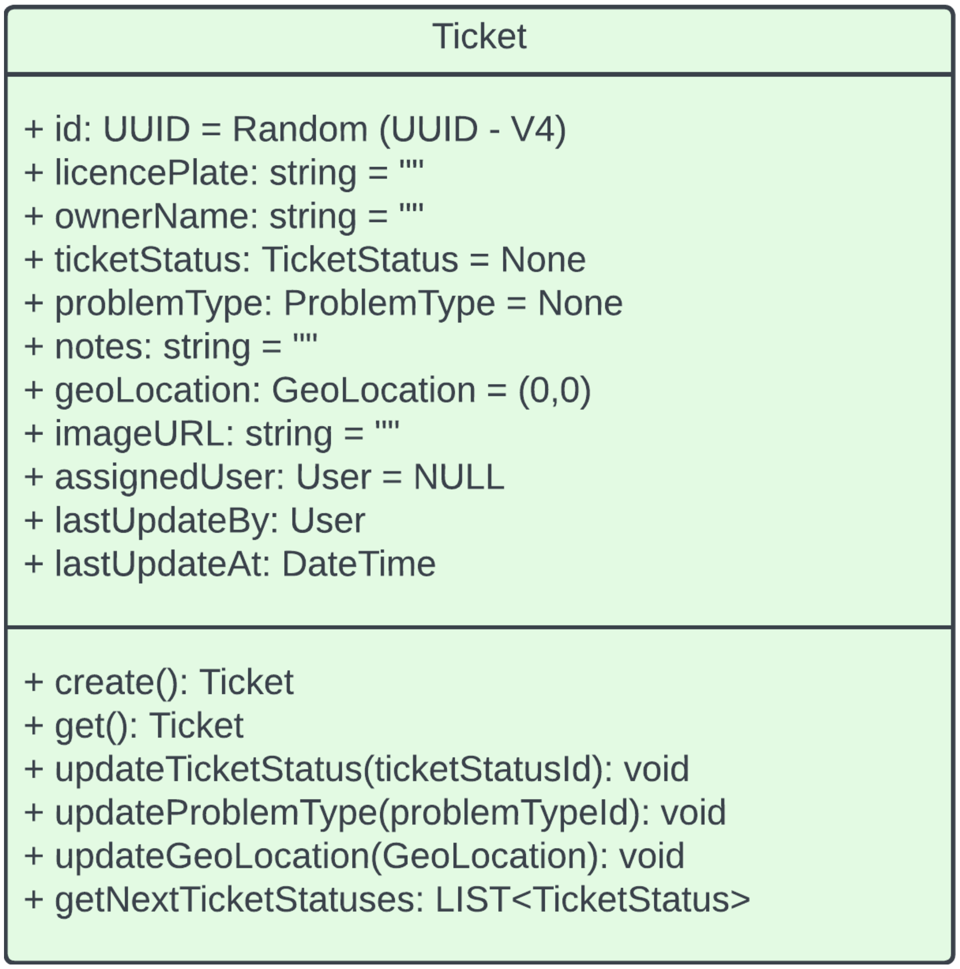
User:



Responsibility
Handles the user and all associated action with this object.

Relationships:
- Main
  - User performs actions upon ticket and other users through main.
- Ticket
  - User can be assigned a ticket. User is responsible for parking or picking up vehicle.
- UserRole
  - User can be valet, admin, or Inactive. Admin has extra privileges and manages valet users. Valet users are responsible for picking up and dropping off vehicles. Inactive users are not currently under employment as a valet, though may be reactivated.

Ticket:



```
Ticket

+ id: UUID = Random (UUID - V4)
+ licencePlate: string = ""
+ ownerName: string = ""
+ ticketStatus: TicketStatus = None
+ problemType: ProblemType = None
+ notes: string = ""
+ geoLocation: GeoLocation = (0,0)
+ imageURL: string = ""
+ assignedUser: User = NULL
+ lastUpdateBy: User
+ lastUpdateAt: DateTime

+ create(): Ticket
+ get(): Ticket
+ updateTicketStatus(ticketStatusId): void
+ updateProblemType(problemTypeId): void
+ updateGeoLocation(GeoLocation): void
+ getNextTicketStatuses: LIST<TicketStatus>
```

Responsibility

This class is responsible for tracking and storage of every vehicle in the system. It encapsulates all properties of a typical exchange that may happen for travelers/valet drivers using this app. This includes important details such as the license plate number, traveler's name, and more. The methods that these properties are bundled together with allow for a simplified interface for creating and updating of a ticket instance. Using the **`create()`** method will return a new Ticket with default properties and it can then be updated with other methods.

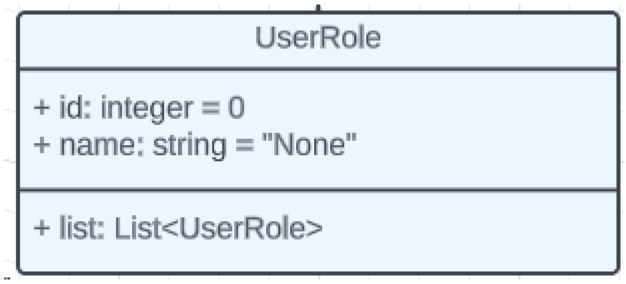
Relationships:

Main (Many to many): The relationship between Ticket and Main is an aggregation because Main contains a list of tickets that still exist without Main.

- **`GeoLocation`** (1 to 1): The relationship between Ticket and **`GeoLocation`** is a composition because every ticket initially has a coordinate location of (0,0) by default and is able to be assigned a single location at any point after that (this is why its 1 to 1 – each ticket has 1 location).

- User (Many to 1): The relationship between Ticket and User has a Many to 1 association. This is because there is no ownership of 1 class over the other, but they are still associated with one another. A user can create any number of tickets, but any given ticket only can be associated to 1 user at a time.
- **`TicketStatus`** (Many to 1): The relationship between Ticket and **`TicketStatus`** is an aggregation because there exists a list of all possible **`TicketStatus`** instances regardless of if they are assigned to a ticket or not. This means that **`TicketStatus`**' existence does not depend on the existence of a ticket, even if there are no tickets created at all. There can be any number of tickets that are assigned a specific instance of **`TicketStatus`**, but each ticket will always have exactly 1 ticket status.
- **`ProblemType`** (Many to 1): The relationship between Ticket and **`ProblemType`** is also an aggregation for the same reason as listed for **`TicketStatus`**. There exists a list of all possible **`ProblemType`** instances regardless of if they are assigned to a ticket or not. This means that **`ProblemType`**'s existence does not depend on the existence of a ticket, even if there are no tickets created at all. There can be any number of tickets that are assigned a specific instance of **`ProblemType`**, but each ticket will always have exactly 1 **`ProblemType`** (even if that problem is None).

UserRole:



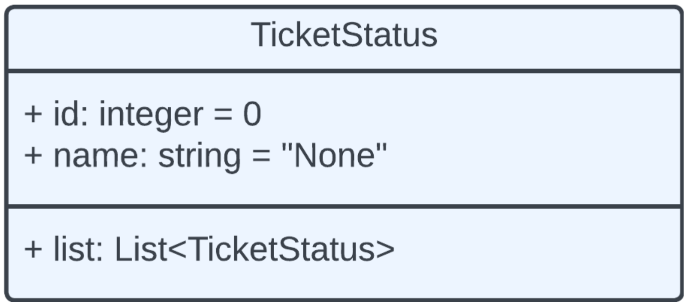
Responsibility

This class works like an enumerated list of possible user roles. There exists 1 instance of each type of user role (valet driver, admin, etc.) and an associated integer value. This implementation provides better modularity (because getting a list of all user roles can be done from 1 class, rather than having a method from another class that gets all user roles).

Relationships:

- User (1 to many): The relationship between **`UserRole`** and User is an aggregation because a User has an "owns" relationship with **`UserRole`**, but **`UserRole`** would still exist without the existence of the User. For **`UserRole`**, there exists an instance of each type of role. Each instance may or may not be associated with a specific user, but each user will have exactly 1 **`UserRole`**.

TicketStatus:



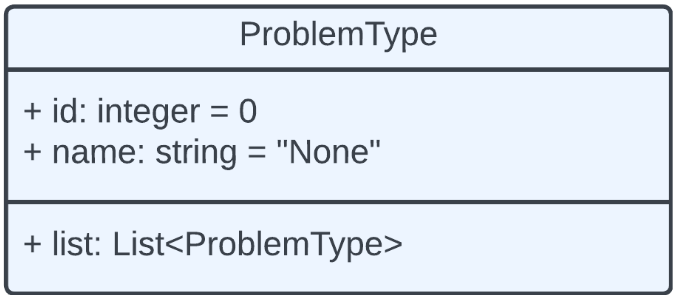
Responsibility

This class holds information about the status of a ticket. It works similarly to the `UserRole` class where there exists an enumerated list of all possible ticket statuses, like dropped off or picked up. The enumerated list of possible ticket statuses exists in the database. The class has 2 attributes: id which stores the integer value of the associated enumerated list entry, and name. The class has one method, list, which returns a list of all `TicketStatus` objects. A Ticket object may have 0 or more `TicketStatus` objects associated with it, but one `TicketStatus` object can only be associated with 1 ticket object.

Relationships:

- Ticket: 0 or more to 1 aggregation

ProblemType:



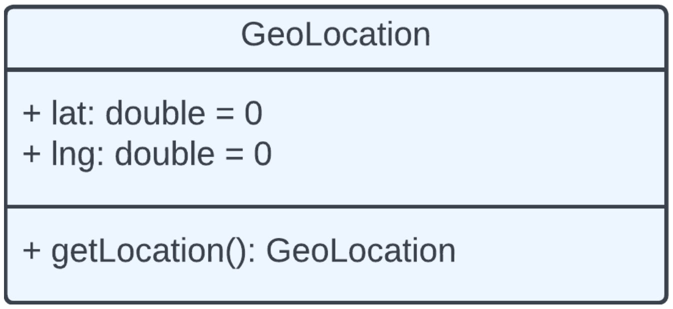
Responsibility:

This class has 2 attributes: id, defaulting to 0 and name, defaulting to none. The class only has one method, list, which generates a list of all possible problem types from an enumerated list. `ProblemType` objects reference data in a `ProblemType` enumerated list stored in the database. 1 `ProblemType` object can be associated with a Ticket object, but a Ticket object can be associated with 0 or more `ProblemType` objects

Relationships:

- Ticket: 0 or more to 1 aggregation

GeoLocation:



Responsibility

This class's role is to store geolocation data. It has 2 attributes, **`lat`** for latitude and **`lng`** for longitude, and 1 method, **`getLocation`**, which retrieves the callers current latitute and longitude. It has a 1 to 1 association relationship with the Main class. It has this relationship because on the app's startup, a **`GeoLocation`** object is automatically created with the default value of 0 for both **`lat`** and **`lng`**. It has a 1 to 1 composition relationship with the Ticket class. This is because each Ticket object will have only 1 **`GeoLocation`** object.

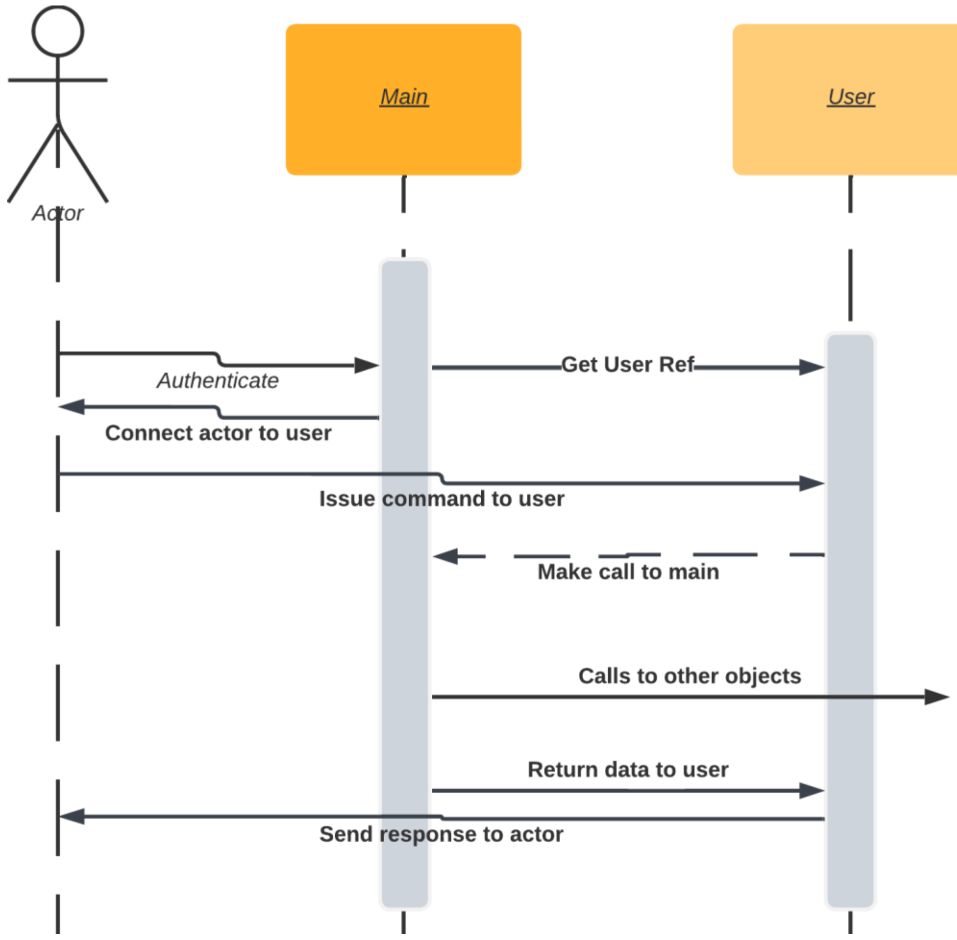
Relationships:

- Main: 1 to 1 association
- Ticket: 1 to 1 composition

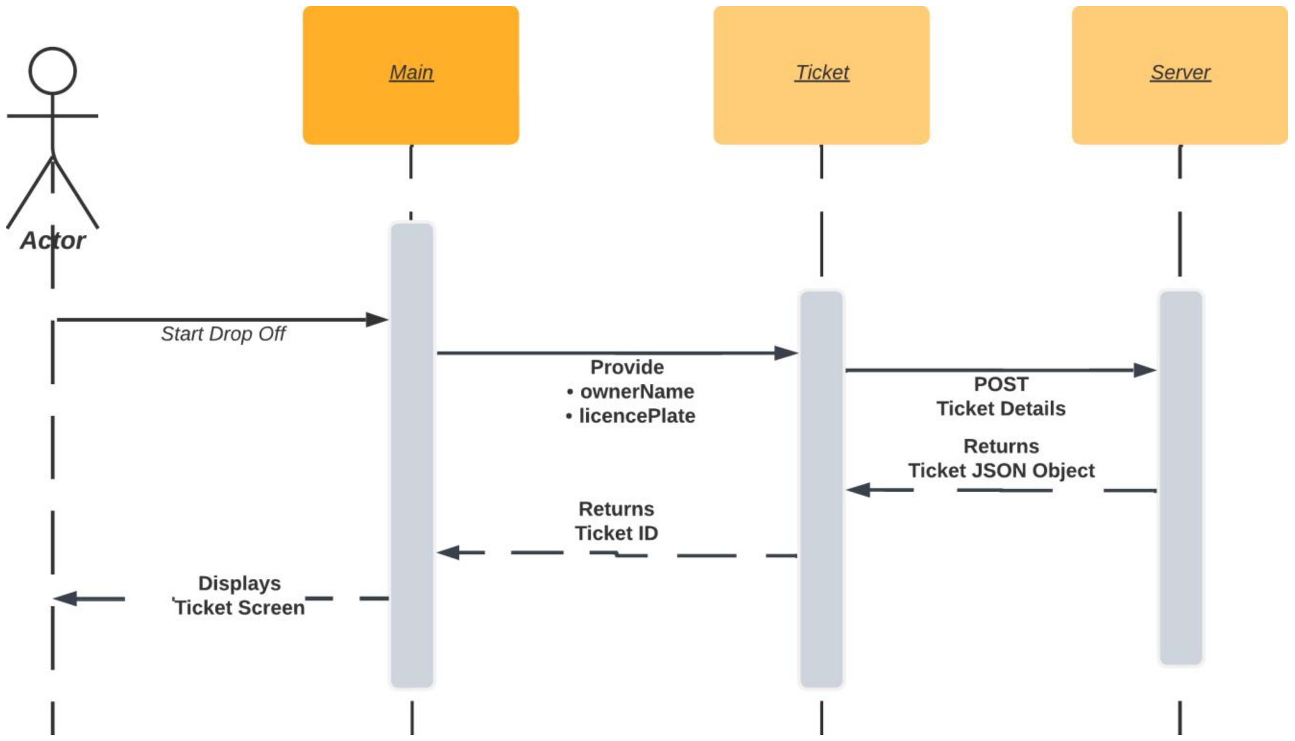
Object Interaction

The following are a sub-set of the object interaction and the sequence diagrams for the Valet Buddy. These diagrams all closely relate to the Unit Testing sequences and phases below.

Authentication Sequence



Ticket Creation Sequence



Unit Testing

Unit testing is designed around testing individual components / classes in isolation to ensure that each part works correctly on its own. Unit tests are the smallest testable parts of an application, typically functions or methods. They are written to ensure that a particular unit of code works as intended. Further, unit testing allows for quick checks and early problem detection at the code level.

Strategy:

The strategy for the Valet Buddy development team is threefold regarding the Unit Testing:

- **Test Early & Often:**
  Integrate unit testing into the early stages of development and continue throughout the project development lifecycle. Meaning that we will try and code "Unit-Tests" while we develop the application and most importantly the classes.
- **Prioritize Critical Paths:**
  We are going to focus on the business logic, user input handling, data validation of the Valet Buddy Mobile application.
- **Implement Flutter Test Package:**
  In Flutter we will be using the **test package** that provides the core framework for writing unit tests, and it also provides additional utilities for testing widgets.
  (https://docs.flutter.dev/cookbook/testing/unit/)

Detailed Unit Test Parameters:

Generally as we build the Valet Buddy mobile application the following unit tests should be considered. This is not a fully comprehensive list.

Ticket

Seq	Method	Parameter	Response	
			Object Type	Details

1	Ticket<<Constructor>> **create()**	"ownerName" = "John Doe"
		"licencePlate" = "123 ABC"
			Ticket	ticket.id = 1 (dynamic)
				ticket.ownerName = "John Doe"
				ticket.licencePlate = "123 ABC"
				ticket.lastUpdateAt = <<Current Date & Time>>
				ticket.titcketStatus = <<TicketStatus.None>>
				ticket.problemType= <<ProblemType.None>>
				ticket.geoLocation = <<GeoLocation(0,0)>>

2	ticket. **get()**	*NONE*	Ticket	ticket.id = 1
				ticket.ownerName = "John Doe"
				ticket.licencePlate = "123 ABC"
				ticket.lastUpdateAt = <<Current Date & Time>>
				ticket.titcketStatus = <<TicketStatus.None>>
				ticket.problemType= <<ProblemType.None>>
				ticket.geoLocation = <<GeoLocation(0,0)>>

3	ticket. **updateTicketStatus()**	"ticketStatusId" = 2	Ticket	ticket.id = 1
				ticket.ownerName = "John Doe"
				ticket.licencePlate = "123 ABC"
				ticket.lastUpdateAt = <<Current Date & Time>>
				ticket.titcketStatus = <<TicketStatus.Parking>>
				ticket.problemType= <<ProblemType.None>>
				ticket.geoLocation = <<GeoLocation(0,0)>>

4	ticket. **updateProblemType()**	"problemTypeId" = 2	Ticket	ticket.id = 1
				ticket.ownerName = "John Doe"
				ticket.licencePlate = "123 ABC"
				ticket.lastUpdateAt = <<Current Date & Time>>
				ticket.titcketStatus = <<TicketStatus.Parking>>
				ticket.problemType= <<ProblemType.Lost>>
				ticket.geoLocation = <<GeoLocation(0,0)>>

5	ticket. **updateGeoLocation()**	"geoLocation" = (12.345, 67.890)	Ticket	ticket.id = 1
				ticket.ownerName = "John Doe"
				ticket.licencePlate = "123 ABC"
				ticket.lastUpdateAt = <<Current Date & Time>>
				ticket.titcketStatus = <<TicketStatus.Parking>>
				ticket.problemType= <<ProblemType.Lost>>
				ticket.geoLocation = <<GeoLocation(12.345, 67.890)>>

6	ticket. **getNextTicketStatuses()**	*NONE*	<<LIST>> TicketStatus	ticketStatus.id = 1 ticketStatus.name = "None" * * *

Main				
Seq	**Method**	**Parameter**	**Response**	
			Object Type	***Details***
1	Main<<Constructor>> **Singelton - Approach** **We only have one isntance of Main for the Application**	*NONE*	Main	main.tickets = LIST<<EMPTY>> main.currentLocation = <<GeoLocation(0,0)>> main.isLoggedIn = FALSE main.currentUser = NULL
2	main **getTickets()**	*NONE*	<<LIST>> Ticket	ticket.id = 1 ticket.ownerName = "John Doe" ticket.licencePlate = "123 ABC" ticket.lastUpdateAt = <<Current Date & Time>> ticket.titcketStatus = <<TicketStatus.Parking>> ticket.problemType= <<ProblemType.None>> ticket.geoLocation = <<GeoLocation(0,0)>> * * *

GeoLocation				
Seq	**Method**	**Parameter**	**Response**	
			Object Type	***Details***
1	GeoLocation<<Object>> **getLocation()**	*NONE*	GeoLocation	geoLocation.lat = 12.345 geoLocation.lng = 67.890

User

Seq	Method	Parameter	Response	
			Object Type	*Details*

1	User<<Constructor>>
	create()
		"email" = "jsmith@mtsu.edu"
		"firstName" = "Jane"
		"lastName" = "Smith"
		"password" = "******"
		"hireDate" = "2023-11-01"
			User	user.id = UUID (Random Set V4)
				ticket.email = "jsmith@mtsu.edu"
				user.firstName = "Jane"
				user.lastName = "Smith"
				user.userRole = <<UserRole.Active>>
				user.hireDate = "2023-11-01"

2	User<<Object>>
	authenicate()
		"email" = "jsmith@mtsu.edu"
		"password" = "******"
			User	user.id = UUID
				ticket.email = "jsmith@mtsu.edu"
				user.firstName = "Jane"
				user.lastName = "Smith"
				user.userRole = <<UserRole.Active>>
				user.hireDate = "2023-11-01"

3	user.
	setPassword()
		"oldPassword" = "******"
		"newPassord" = "******"
			BOOLEAN	success = "True / False"
				message = "Error Message about Update"

4	user.
	update()
		"firstName" = "Janey"
		"lastName" = "Smither"
		"hireDate" = "2023-11-15"
			User	user.id = UUID
				ticket.email = "jsmith@mtsu.edu"
				user.firstName = "Janey"
				user.lastName = "Smither"
				user.userRole = <<UserRole.Active>>
				user.hireDate = "2023-11-15"

5	user.
	delete()
		NONE	User	user.id = UUID
				ticket.email = "jsmith@mtsu.edu"
				user.firstName = "Janey"
				user.lastName = "Smither"
				user.userRole = <<UserRole.Deleted>>
				user.hireDate = "2023-11-15"

Integration Testing

Integration testing is in the Valet Buddy’s application domain and is about ensuring that the above-mentioned components / classes work together as expected when they are integrated into a larger system (Valet Buddy Mobile App). Furthermore, integration testing ensures that the system as a whole functions correctly and as expected from a user and application governance standpoint.

Strategy:

Our integration strategy will focus on the follow three (3) main areas to meet the guidelines of this simple mobile application:

- **Test Across Devices:** Include tests that account for different screen sizes, OS versions (Android and iOS Apple).
- **Peer Reviews**: Incorporate code reviews to ensure tests are readable, maintainable, and effective among the developers and teams.
- **Performance Testing:** We will strive to include performance tests as part of your integration strategy to monitor the Valet Buddy’s mobile app's performance and ensure it meets the required benchmarks for speed and error messaging.

Detailed:

While discussing the strategy for testing the interactions between classes and components. These are listed below in the following pages:

- **Sequence Diagrams:** Utilize sequence diagrams that we mentioned above or detailed flowcharts to understand and verify the intended interactions and ensure that tests cover these sequences.
- **Interface Testing:** We will be focusing on the points where classes and components interface with each other, ensuring that data is correctly passed and received among the class instances.
- **Ticket State Testing:** Verify that the Valet Buddy maintains the correct states in the **`<<Ticket>>`** instance throughout the interaction process, especially when dealing with types of stateful components that are updated with the app.

Meeting Minutes & Notes

The below are simple meeting minutes of the discussions we as a group had during the development of this "Design Document" document.

2023-10-30 – In Class Meeting:

Started basic setup/outline of this document and initial group discussions on how to complete this assignment. Wrote an Introduction and explained each of the Object-Oriented Principles.

2023-11-01 – In Class Meeting:

Began working on a LucidChart diagram for the Class Diagram section of this document. We had a long discussion on how to proceeded and asked the professor to help explain some sections.

2023-11-02 – Friday Morning Meeting (zoom):

We spent as a group a while working on the Class Diagram. There are still a lot of uncertainties that we are unsure of. We had to split up some of the work to make sure that we can get this done by the deadline this evening. Richard is working on the unit and integration test cases, Jordan & Patrick on the Class Diagram details, and Ian on the Sequence work.

A lot of discussion took place on if we need a UML Class Diagram for the Serve side of the system. We opted to see if Richard has time later on today to try and create one for this document, but for now we need to focus on getting the basics done for submission.

Set up another meeting for Saturday morning to start creating the PowerPoint presentation on this document... phew... Will be working on the PowerPoint tomorrow.