Project Name: Robert's Mountain Shop Management

1.0 Project Description

1.1 Purpose and Goal

The purpose of this project is to help a customer rent snow-and-mountain items from Robert's Mountain Shop. Right now there is no system in place to track current and outstanding rentals and/or rent items from the shop other than by hand-and-paper methods. The goal of this project is to create an application that will better facilitate the renting of various items to a customer. Ideally, the customer will be able to use a simple and easy-to-understand interface that will let them choose items to rent and checkout. After they check out, their order will be processed and recorded so that a employee can view outstanding or current orders.

1.2 Benefits

The benefits of this project will be to the employee and customer. Instead of having to wait on an employee describe each item, the customer will have all the information they need about any of the items at their fingertips. Likewise, the employee will have an easier time facilitating a transaction as the only work that will need to be done by them will be taking payment via an out-of-scope system.

1.3 Implementation

Architecture: The software architectural design that was used was MVC (Model-View-Controller). The goal of this architecture is to separate the input, processing and output of the application. The model used in our application will be that of a sqlite3 database. In the project files this database is listed as *database.db*. The controller's in our application are all of the mapped functions in *main.py*. Finally, the view in our application are the html files in the *template* folder. These views will format the processed data from the controllers and render them onto the html pages.

Frameworks: This application is using a micro-framework called Flask. This framework includes many external packages that can help incorporate many functionalities that micro-frameworks usually lack such as account handling and database abstraction via object-relational mapping.

Languages and Object-Oriented Programming: The languages used in this application will be a mix of Python, Jinja2 and HTML. Given that Python doesn't have things such as private variable descriptors or interfaces that you can implement, in order to design an object-oriented-programming application is a little different.

The way I used hidden/private variables is by declaring a private variable with "_" which would look something like: "_variableName". In the python engine if you try to access that variable outside of the class (i.e., "classInstance._variableName") it would fail and say that the variable is not defined within 'classInstance'.

Abstract classes, interfaces and other usual concepts that are included in OOP projects are used in different ways in python or not implemented at all. The concepts are different and in my project you will see that I still followed good practices even though no adhering to Java or C# style object-oriented programming.

2.0 Features

2.1 Features That Were Included

Features That Were Included				
Use Case ID	Use Case Name	Primary Actor	Complexity	Priority
1	Create an Account	Generic User	1	1
2	Login to Account	Generic User	1	1
3	View Rentals	Customer	2	1
4	View Current Rentals	Customer	3	2
5	Checkout	Customer	2	1

User Stories

Use Case 1:

As a generic user I want to be able to create an account so that I can have an account to use.

Use Case 2:

As a generic user I want to be able to login to my account so that I can access the applications dashboard.

Use Case 3:

As a customer I want to view rentals that are available to rent so that I can choose which ones I want to rent.

Use Case 4:

As a customer, I want to view the current rentals I've selected so that I can know how much my total will be.

Use Case 5

As a customer I want to be able to checkout so that I can get the items I've chosen to rent.

2.2 Features That Were Not Included

Features That Were Not Included				
Use Case ID	Use Case Name	Primary Actor	Complexity	Priority
6	Return Rentals	Customer	2	3
7	View outstanding Rentals	Employee	3	2
8	Report a Lost Item	Customer	3	3
9	Include Discounts	Customer	2	3
10	Delete Account	Generic User	2	3

User Stories

Use Case 6:

As a customer I want to be able to return my rentals so that I can complete my contract with the mountain shop.

Use Case 7:

As an employee I want to be able to see all outstanding orders to know how many orders are currently outstanding and what items are currently rented.

Use Case 8:

As a customer I want to report lost items so that if I do lose an item I will be able to notify the mountain shop.

Use Case 9:

As a customer I want to enter a discount code during checkout to receive a discount on my rental order.

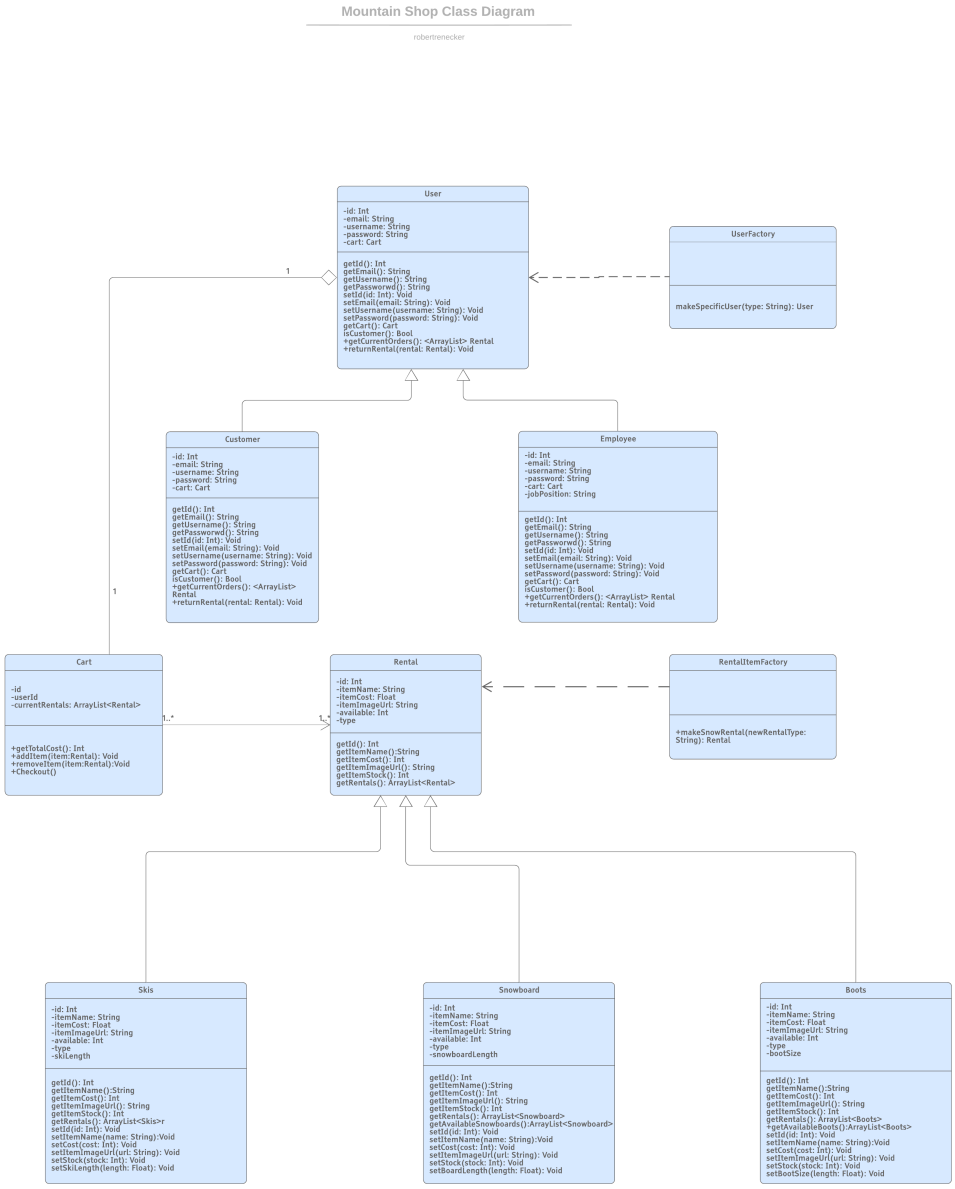
Use Case 10:

As a generic user I want to delete my account so that I can no longer have it contained as a user in the database.

3.0 Class Diagram

In comparison to my last class diagram I think a lot has changed. First of all, the top-down architecture is still somewhat similar but I now have come to realize how important composition is over inheritance. Instead of my first few class diagrams where a Rental was a part of a user class, it is now no longer a part of the user class and is instead only connected to Carts. A user will likewise use composition to have an instance of a cart and then let all the desired functionality of the cart and rental items be done by the cart.

Next, even though there isn't much *functionality* as a whole in my application, I am noticing how much more functionality and things that these classes can do is shown in the class diagram than my first few. I would say the majority of my functionality other than getters and setters is now taking place in the Cart class.



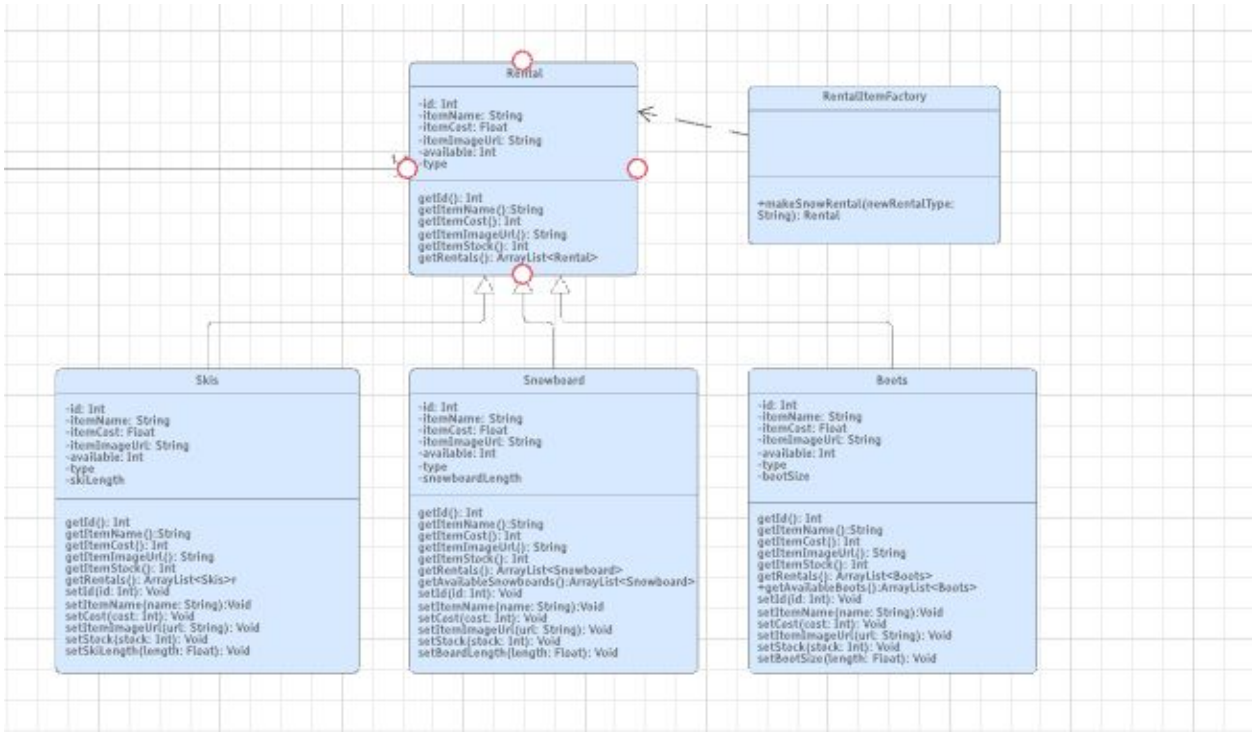
4.0 Design Patterns

4.1 Factory Method Design Pattern

What I ended up using for a design pattern was the Factory Method Design Pattern. I chose to use this pattern because I will end up implementing a use case for the employee where they can add new rental items to their stock at run time. This pattern helps with use cases like that because it will then be able to handle the creation of specific subclasses at run time.

Rental Item Factory

The cases where I used it so far was just to encapsulate object creation. In my ‘push_snowitems_to_db.py’ file I created objects that ranged from snowboards, skis and boots. These were all children of the Rental class and I was able to create them using the RentalItemFactory class.



In the class diagram above we can see that there is a class called ‘RentalItemFactory’. This has a function called makeSnowRental with a input of the type of rental it will create. It will then return that concrete rental specified by type and return that to the original controller attempting to create the object.

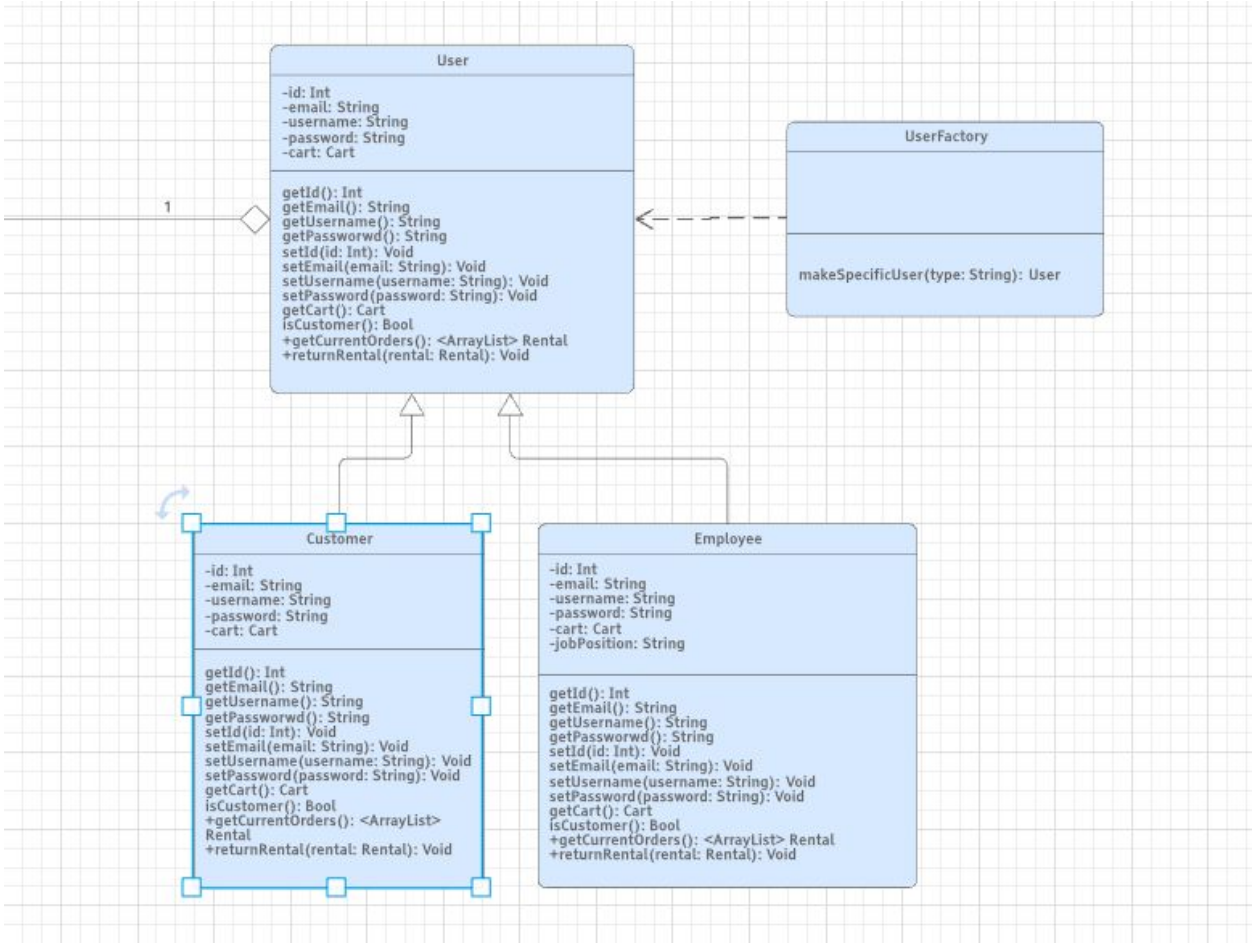
To see the code and how I implemented those classes, I have below a RentalItemFactory class which will be created with the attributes of a Rental item (the attributes which all rental items have), and then let the use create a specific rental item such as Snowboard given a specification for the type “snowboard”. I also included below a Snowboard class which is a child of the Rental class.

```
class RentalItemFactory():
    def __init__(self, id, itemName, itemCost, itemImageUrl, stock):
        self._id = id
        self._itemName = itemName
        self._itemCost = itemCost
        self._itemImageUrl = itemImageUrl
        self._stock = stock
    def makeSnowRental(self,newRentalType):
        if newRentalType == "Ski":
            return Skis(self._id, self._itemName, self._itemCost, self._itemImageUrl, self._stock);
        elif newRentalType == "Snowboard":
            return Snowboard(self._id, self._itemName, self._itemCost, self._itemImageUrl, self._stock);
        elif newRentalType == "Boots":
            return Boots(self._id, self._itemName, self._itemCost, self._itemImageUrl, self._stock)
        else:
            return None
```

```python
class Snowboard(Rental, db.Model):
    _boardLength = db.Column(db.Float)
    __mapper_args__ = {
        'polymorphic_identity': 'snowboard'
    }
    def __init__(self,item_id, item_name, item_cost, item_image_url,stock):
        self._id = item_id
        self._itemName = item_name
        self._itemCost = item_cost
        self._itemImageUrl = item_image_url
        self._stock = stock
        self._boardLength = 180

    def getId(self):
        return self._id

    def getItemName(self):
        return self._itemName

    def getItemCost(self):
        return self._itemCost

    def getItemImageUrl(self):
        return self._itemImageUrl

    def getStock(self):
        return self._Stock

    def getBoardLength(self):
        return self._boardLength

    def getRentals():
        available_rentals = Snowboard.query.filter(Snowboard._stock>=1)
        return available_rentals

    def setId(self, id):
        self._id = id

    def setItemName(self, name):
        self._itemName = name

    def setCost(self, cost):
        self._itemCost = cost

    def setItemImageUrl(self, url):
        self._itemImageUrl = url

    def setItemStock(self, stock):
        self._stock = stock

    def setBoardLength(self, length):
        self._boardLength = length
```

User Factory

The other case where I implemented the Factory Pattern was in the creation of my user accounts. I wanted to have different accounts be able to do different things. What I have so far is basic information that I think is needed for each user and for my application at this current time.



In the class diagram above we can see just like the Rental Item Factory that the UserFactory has a function that is able to return a concrete user class. The concrete user class (at this current stage) will be either a type Customer or Employee.

The code for my user classes is a little cleaner than that of the rental item classes. Below I included a UserFactory, a User class (the parent), and a Customer class (child of parent). When creating a user, the client will just use the UserFactory and if given a type the makeSpecificUser function will return an instantiated class of that type. If it is a Employee, a job position will be included with the instantiation of a Employee type user.

```python
class UserFactory():
    def makeSpecificUser(self, type, username, email, password, jobPosition=None):
        if type=="customer":
            return Customer(username, email, password)
        elif type=="employee":
            return Employee(username, email, password, jobPosition)
```

```python
class User(UserMixin, db.Model):
    static_id = 1
    id = db.Column(db.Integer, primary_key=True)
    _username = db.Column(db.String(15), unique=True)
    _email = db.Column(db.String(50), unique=True)
    _password = db.Column(db.String(80))
    type = db.Column(db.String(32))

    cart = db.relationship("Cart", backref='user', lazy=True, uselist=False)



    def __init__(self,username, email, password):
        self.id = User.static_id
        self._username = username;
        self._email = email;
        self._password = password;

        User.static_id += 1

    __mapper_args__ = {
        'polymorphic_identity': 'user',
        'polymorphic_on': type,
    }

    def getCart(self):
        return self.cart;
    def getUsername(self):
        return self._username
    def getEmail(self):
        return self._email
    def getId(self):
        return self.id
    def getPassword(self):
        return self._password
    def getCurrentOrders(self):
        return self.currentOrders

    def setUsername(self, username):
        self._username = username
    def setEmail(self, email):
        self._email = email
    def setId(self, id):
        self.id = id
    def setPassword(self, password):
        self._password = password
    def setCart(self):
        self.cart = Cart(user = self)

    def isCustomer(self):
        pass
```

```
class Customer(User):
    __mapper_args__ = {
        'polymorphic_identity': 'customer'
    }

    def __init__(self,username, email, password):
        super().__init__(username, email, password)
    def isCustomer(self):
        return True
```

5.0 What I Learned

Throughout this project from start to finish I have been confused, sad, happy, sometimes lucky to find easy solutions, but most importantly, glad that i've overcome difficult obstacles. I think one of the hardest problems for me was understanding how to brainstorm and properly analyse and design a project at the beginning of one. I spent many hours trying to fix problems that in retrospect, if I had been better prepared, they would of never have occured. Next time, or if I decide to continue working on this project, I would like to involve myself more in the design steps of writing specific user stories so that I can try and find relevant design patterns to help accomplish satisfying those stories. I also would like to better facilitate my organization of the system architecture by writing out detailed class diagrams and sequence diagrams. This process I believe will help any one developer better understand the project they're undertaking and help them plan accordingly for solutions they hope to implement.

From this project specifically, I have come to better grips with understanding the MVC architecture, the Flask microframework and OOP programming within it. I have come to better understand why separating the input, processing and output of data is important. One of the benefits I felt from using MVC architecture was that I could modify one part of it and it wouldn't affect the entire architecture of my application. This helped with debugging and adding new things that I wanted to test out. Another great benefit I noticed by using MVC was that the model doesn't return formatted data. I can take that data to multiple views and use it however I want which makes it easy to use the data with any interface.

Overall, the process of designing an object-oriented-programming project is still a difficult one for me but, I believe that through the many hours I put into this project that I have gained a great deal of knowledge. I have learned of the importance of how objects function rather what they have as attributes, how and why you don't let the client know of the implementations of classes and the processes they carry out and lastly how to structure classes so they have composition over inheritance.