**Winter Semester 2019/20**

Project Plan of

**Application Design**

**Muhammad Ali**

**00732065**

**Prof.** **Marcus Barkowsky**

**Abstract requirements definition**

* **A simple copy of the abstract requirements definition above (direct copy-and-paste for your documentation and later reference)**

The context of the project is traffic management in a city (>100.000 inhabitants).

You are going to design an application that provides an innovative approach to one of the many problems in the traffic management. You choose the topic of your specific project, for example managing the time schedule of public transportation.

While you are free to choose your specific topic, your project shall fulfill the following requirements:

* The estimated effort for the implementation of the project is 20 man-years and the delay is less than a year, i.e. you can work with at least 20 people in parallel.
* You need more complex software architecture than a monolithic program.
* At least some part of the project requires a Human-Machine Interface that can be implemented with a touch-screen GUI interface following the guidelines on utility, usability, UX and QoE.
* At least one of project's entities are controlled by two input sources (i.e. the arrival time of the bus depends both on the traffic simulation (source 1) and the preplanned schedule (source 2)).
* Your project is suitable for implementing one set of design patterns. The sets are detailed below.

Choose one of the following sets of design patterns. You have to implement each design pattern at least once in your project.

* Set 1: Composite, Memento, Builder, Adapter, Observer
* Set 2: Visitor, Abstract Factory, Interpreter, Singleton, Interceptor
* Set 3: Chain of Responsibility, Decorator, Factory, Proxy, Flyweight
* Set 4: Active Object, Monitor Object, Composite, Decorator, Strategy
* **The explanation of your interpretation of the abstract requirements, i.e. explaining which scenario you implement and where each of the abstract requirements is implemented. (0.5-1-page A4)**

A parking lot or vehicle park is a designated paved area intended for car parking. Parking a lot are an attribute of each and every city and suburban region in most countries where cars are a major mode of transportation. Shopping malls, sports activities stadiums, mega-churches and similar area additionally. I will be designing a parking Management System. Finding parking, advance parking, free parking weekly and monthly Etc. This software is fulfilling UX and device QoE requirements.

There are six main steps for user:

* Step #1. Searching: The user locates the car using GPS or enters the necessary address.
* Step #2. Comparing: The app offers quite a few versions of parking spots. The user can filter them via price, vehicle types or distance to locate the best match.
* Step #3. Booking: The driver reserves the chosen parking area and gets an access code.
* Step #4. Paying: The user can pay by credit card in advance or in cash later on exit gate.
* Step #5. Driving: The parking app offers driving directions to the parking lot.
* Step #6. Parking: The driver enters the parking lot and parks the car.

User can also advance booking through software, Call, online chat and email when he or she on the way.

* **A concise explanation of the particular software architecture (i.e. n-tier architecture, storage location, the set of design patterns chosen)?**

**MVC** is a software design pattern that Development of the application becomes faster & easy our “Parking management system “. The **Model-View-Controller (MVC)** framework is an architectural pattern that separates an application into three main logical components Model, View, and Controller. Hence the abbreviation MVC. Each architecture component is built to handle specific development aspect of an application. As my experience MVC is the best software architecture it’s easy for multiple developers to collaborate and work together & Easier to update the application.

* Model: It consists of all the information and its related logic
* View: Present statistics to the person or handles user interaction
* Controller: An interface between Model and View components

### View (Presentation Layer)

A View is that section of the software that represents the presentation of data. Presentation layer is also known as Client Layer. It is a user display where an end user could see text boxes and button to enter location for search parking lot. After putting the location there then presentation layer is to communicate with application layer. This layer passes the information which is given by the user in terms of input data from keypad or touch screen to the application layer. Handles data presentation dynamically rendered.

**Model (Application Layer)**

Application layer is also known as Business logic layer which is additionally known as logical layer. As per my “Parking management system” software example, once user click on the login button, Application layer interacts with Database layer and send required information to the presentation layer. It controls an application’s functionality by performing detailed processing. This layer acts as a go-between the presentation and the Database layer. Complete business logic will be written in this layer. Handles data logic Interacts with database.

**Controller (Data layer)**

The data is stored in Data Layer. Application layer communicates with Database layer to retrieve the data. It contains methods that connects the database and performs required action Example: insert, update, erase etc. handles request flow, in no way handles data logic.

* **A detailed explanation of the usefulness of each design pattern (1 page A4)**

1. **Memento Design Pattern:**

The Memento pattern is typically used to store the historical state of an object and, therefore, allow previous states to be restored at any time. The pattern shares many similarities with the command pattern.

The **memento design pattern** has three key components:

* **Memento** – Simple object that contains basic state storage and retrieval capabilities.
* **Originator** – Gets and sets values of Mementoes. Also, creates new Mementoes and assigns current values to them.
* **Caretaker** – Holds a collection that contains all previous Mementoes. Can also store and retrieve Mementoes.

Problem Solving:

* Need to restore an object back to its previous state.
* Its supports Unlimited time of “undo” and “redo” /” rollback’ operations

**Applicability:**

For example, Consider the “Parking lot “software we implementation on Payment class we frequently exchange or unchanged our fee according the days, holidays and other events days. On weekend we keep low price and rather than weekdays rate. If the rate is set is 3 € first hour, then 2 € for Second hours, you favor to change rate on holiday or weekdays 2 € first hour, then 1.50 € for Second hours! The act of changing that rate state from 3 to 2 is a simple form of the memento pattern. In which we’re changing the *state* of an object (the rate, in this case) to a new value. Critically, we can also *revert* that change at any time, going back to any rate state value we had previously.   
For the weeks day we set **Undo** Operation on weekday’s charge € 3 per hours this process will be incremental, meaning we go parking rates from Euro €3 to €2 to €1, and so on as per hours. Yet, on weekend set **Redo** Operation for events or any National holidays, we can explicitly choose rates to jump to immediately previous rates, making the change from €3 to €1.5 or €1 in a single step. Regardless, both of these techniques are commonly used in **memento design** **pattern** implementations. **Memento component** captures every state and keep store internally, so **originator component** uses the memento components for going from current state to previous state to perform rollback operation anytime. And last **caretaker component** it keeps data safely and ensure information don’t be corrupt. We can access our previous state without any problem or data exploitation.

**Builder Design Pattern**

Builder pattern builds a complex object the usage of simple objects and using a step by step approach. This kind of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. Encapsulate code for construction (related with class and object creation) and representation (user interface related).

A Builder classification builds the final object step by step. This builder is independent of different objects.

The **Builder design pattern** has four key components:

* **Builder:**  is the base classification which all the **concretebuilder** cases extend.
* **Director:**  holds a reference to a **Builder Director** directs i.e. is responsible for creation of the product using the interface provided to it by builder.
* Each **ConcreteBuilder** is responsible for a variant of the product.

While **buildPart ()** technique triggers the advent of sub-part, the **get result ()** method is furnished to fetch the last assembled products.

* **Product**: object represents the complex final product which is being constructed.

**Applicability:**

When we want system should have Separate object constructing form its representation.We have considered a Parking administration system where a different vehicle in and out at the same time. In the parking lot Vehicles could be either an Electric car, non- electric car, lorry, heavy truck Etc. All customers and all staff members will get notification in different ways suppose Admin “SMS/Email”, Operation Manager “SMS/Email”, Customer “SMS”, Parking Attendant “SMS” will be notifying after the vehicle Enter /Exit the Parking.

Here we need to create **NotifyCreator** for vehicle leaving**.** Notify Creator class get the input as park the vehicle in parking lot from client. It has occurrence of conceptual class notify creator. It uses the appropriate interface of **NotificationBuilder** class to create the components of objects it requires via logic in Construct () method. The concrete class email **Notification Builder** manufactures the parts of object for personal notification through implementation of method notification administrator () and notice Manager (); and gathgers it as object of Notification Builder class comparatively, the concrete class **SmsNotificationBuilder** tell the Notification of object for SMS notification builder through implementation of method **NotificationCustomer();** and NotificationParkingAttendant() and assembles it as object of SmsNotificationBuilder class. Client get the final object through method get **Notification Builder ().**

The Builder Pattern is similar to the Abstract Factory Pattern in that both patterns are designed for creating complex objects that are composed of other objects. What makes the Builder Pattern distinct is that the builder not only provides the methods for building a complex object, it also holds the representation of the entire complex object itself.

**Composite Design Pattern**

Composite pattern lets clients treat the individual objects in a uniform manner.

You want to represent part-whole hierarchies of objects.

you need customer to have the option to overlook distinction between organizations of items and individual objects.

The Composite Design Pattern has four components:

• **Component:** (Employee) declares the interface for objects in the composition.  
 - Implements default behavior for the interface common to all classes, as appropriate.  
 - declares an interface for accessing and managing its child components  
• **Leaf**: (Developer, Maintainer, Business Analysis, etc.)  
 - represents leaf objects in the composition. A leaf has no children.  
 - defines behavior for primitive objects in the composition.  
• **Composite:** (Manager) defines behavior for components having children.  
 - stores child components.  
 - implements child-related operations in the Component interface.  
• **Client:**  manipulates objects in the composition through the Component interface.

What makes the Composite pattern one of the most delightful is the intensity of recursion. I can explain this with the same organization of my Parking management system example. When we want to find the total salary paid to all employees of the organization. It is nothing but the salary of CEO + the salary paid to all the departments. What is the salary of a department? It is the salary of the department head + the salary of all projects. What is the total salary of a project? It is the salary of the project manager + the salary of all the project members. In short, the salary of anything is the salary of self + the salary of all its subgroups. In our small organization, there are 5 employees. At top position, there is 1 general manager under general manager, there are Four employees, one is the manager and other is developer and further manager has four employee developers, Business Analysis, Maintenance, And Penetration working under him. We need to print name and salary of all employees from top to bottom. The **Component** interface defines the interface that all objects in the composed system need to use, whether they are **leafs** (simple objects) or **compositions**. However, this is usually implemented as an abstract class providing some default behavior for the add, remove and getChild methods.

The **Leaf**has no children, and accordingly just need to execute the activity technique. The **Composite** needs to do more, as it also contains components. The composite will more than likely need to implement the activity strategy, which is considered as a Leaf-related activity. Sometimes this may not make sense for a composite to implement.  Usually, the Composite will implement methods by delegating to the children.

The **Client**simply uses the Component interface to manipulate the objects.

**Observer Design Pattern**

An observer object can register or unregister from subject at any point of time. It helps in making the objects **loosely coupled**. The observer pattern has four participants.

* **Subject**(BookingNews)– [interface or abstract class](https://howtodoinjava.com/oops/exploring-interfaces-and-abstract-classes-in-java/) defining the operations for attaching and de-attaching observers to the subject.
* **ConcreteSubject** (ConcreteBookingNews) – concrete Subject class. It maintains the state of the object and when a change in the state occurs it notifies the attached Observers.
* **Observer** (ParkingUpdate) – interface or abstract class defining the operations to be used to notify this object.
* **ConcreteObserver** (EmailSubscriber), (EmailSubscriber) – concrete observer implementations.

**Applicability:**

The flow is very simple to understand. Application makes the concrete subject (ConcreteBookingNews) object. All concrete observers (EmailSubscriber), (SmsSubscriber) register themselves to be notified for any further update in the state of subject. As soon as the state of subject changes (BookingNews), subject notifies all the registered observers and the observers can access the updated state and act accordingly. They will get notify frequently form of our special offer or upcoming booking parking. Main reason to implement this observer is handling one-to-many dependency objects so that when one objects changes state, all its dependents are notified and updated automatically. Observer must register to subject to get notified for change instead of observing or asking to the subject again and again.

**Adapter Design Pattern**

One of the structural design patterns is the adapter design pattern and it is used so that two unrelated interfaces can work together. An adapter is called the object that combines certain different interfaces.

An adapter pattern is also known as **Wrapper pattern** as well. Adapter Design is very useful for system integration when some other existing components have to be adapted by the existing system without source code modifications.

The observer pattern has four participants.

•**Target** (SmsNotification)  
 defines the domain-specific interface that Client uses.  
• **Client** (SpecialOfferAlert)  
 collaborates with objects conforming to the Target interface.  
• **Adaptec** (SendSms)  
 defines an existing interface that needs adapting.  
• **Adapter** (SMS Notification)  
 adapts the interface of Adaptec to the Target interface

**Applicability:**

SMS Alert is one of the most powerful means of distributing information to your customer, making it the most efficient way of reaching your customers compared to email marketing, phone calls, or online banners. Email and phone calls have a higher spamming rate compared to relatively low messages. We use laptop, central SMS or any d4evice to send bulk, android cell phone, and serial port to mobile gsm modem. Here the client simply calls the serial port () adapter process. Adapter calls Gsm Modem () on the adapter and returns the expected result for the device. Inside the Adapter, the serial port methods can connect all interface differences and sends SMS.

* **The proposed work for each of the 20 people that will work in your project, i.e. the proof that your software design allows for the separation of concerns approach (1page A4)**

This aims at delivering a high quality programme that meets or exceeds customer expectations, operates effectively and efficiently in the current and planned information technology infrastructure, and is easy to maintain and improve cost effectively. I am going to divide my work between 20 people with Systematic process building software that ensures the quality and correctness of the software built.

|  |  |  |
| --- | --- | --- |
| **S. No** | **Job Title** | **Quantity** |
|  | Database | 3 |
|  | Software Design creation UX | 3 |
|  | User interface implementation | 4 |
|  | Backend implementation | 7 |
|  | Software quality assurance | 3 |
|  | **Total Persons** | 20 |

This is what I can think hierarchically the roles are organized in my company for this project.  
  
**Architect** - Who completely oversees the project?? . All the initial ideas and requirements are finalized by this person.  
  
**Designer** - Designs the complete project from how the data would flow and how the control is handled by different modules at various levels of project.  
  
**Software development engineer** - Codes the complete or partial modules of project divided among different teams or teammates.  
  
**Verification engineer** - Verifies the data and control path on a conceptual product or simulation.  
  
**Validation engineer** - Validates the product after its out and tries to see to it if there are bugs in the product. Goes through changes and again to verification engineer to design engineer and then back to validation engineer to resolve the bugs.  
  
**Quality assurance engineer** - Confirms all the features of the product to be working and finalized just to see that every data and control path in the project behaves as prescribed.

* **The identification of at least three use-cases for your scenario that demonstrate the usage of your software architecture framework (i.e. the design patterns usefulness) with a short explanation (0.5 pages A4).**

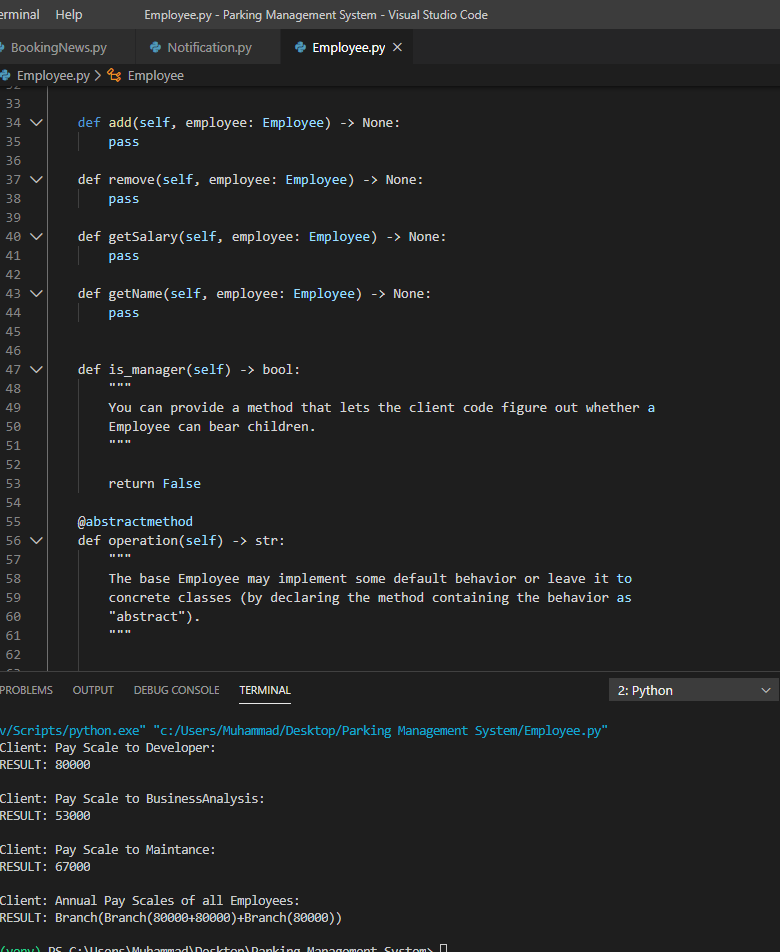
A use is based on a scenario which demonstrates how an actor will interact with the system to perform or achieve a specific goal. The notification implements first of the Builder design pattern. Whenever the customer enters the parking lot, after getting the parking ticket he/ she will receive SMS / mail information about the parking slot Example a1, a3, d7 etc. It also filled parking slot time period. Parking attendant, Op Manager and Admin get the same notification as parking attendant. Builder design pattern is applied to create complex object composed of other objects. So, in this situation this design pattern is the best fit. As mentioned above, the pattern of observer design is applied on a class of bookingNews. The **Observer patterns** a one-to-many relationship so that when one object change state, the other are notified and update automatically. When the customer parks online reserved for regular, weekly and monthly, he will receive periodic notification of the time slot number and period and our new deal for students and senior citizen. **Composite design pattern** is applied in the Employee class. We have introduced this design pattern that requirements to find the total salary received by the organization’s employee. So composite design pattern allows us to treat individual objects and compositions object uniformly. 1-to-many "has a" up the "is a" hierarchy

* **An explanation how you designed your GUI and where and why you would take into account UX and QoE (0.5-page A4 plus scribbles/wireframes etc.)**

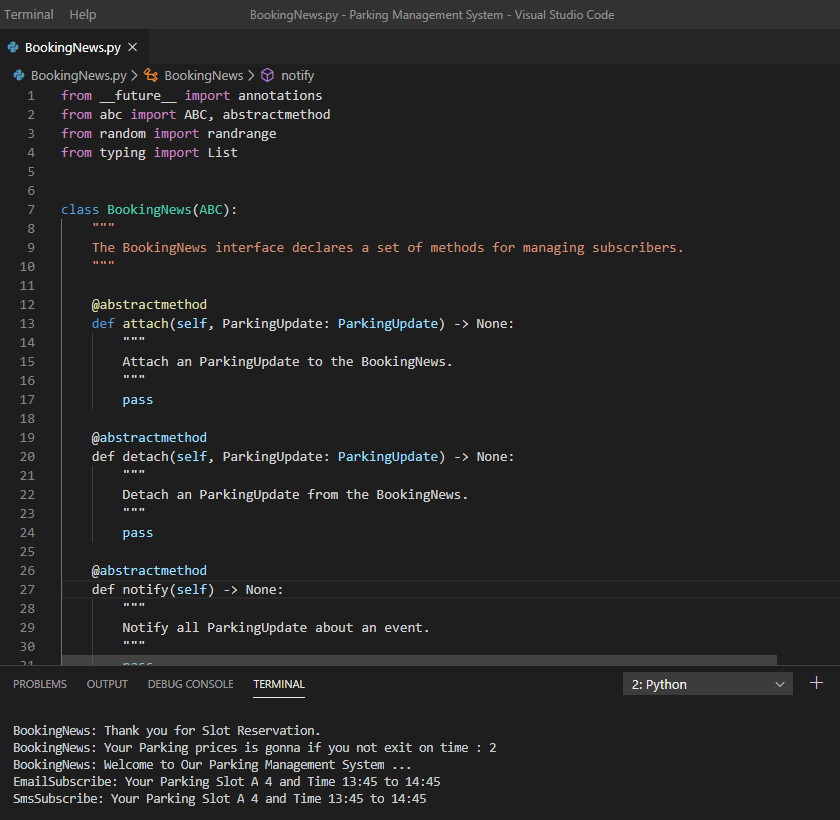
Graphical user interfaces (GUIs) surround us in our daily lives – with touch screens on our Parking Management system, washing machines, thermostats and coffee makers at home and touch controls on copying machines and industrial devices at work. A GUI’s are combination of graphical and textual interaction that uses buttons, menus, message boxes, etc. So, in Parking Management I provided a simple and effective means of connecting systems, collecting data and presenting information in a meaningful format. A good design leads to increase productivity and gain. Icon replaces text to represent object or action, simplifying and making interaction are much easier. Usability is the major concern in the information systems like this envisioned system. The usability will be taken into account when we have to implement the interface of the system. While creating the mockups for the user interface of the system the quality of experience will be used to determine that how easy and fast it will be for the users to understand and use the system. The quality of experience provides a roadmap in implementation of the quality systems. To take usability into the account the Gestalt laws will be implemented to develop the user interface of the system. Following are the laws presented by Gestalt to make the user interface better and easy to understand.

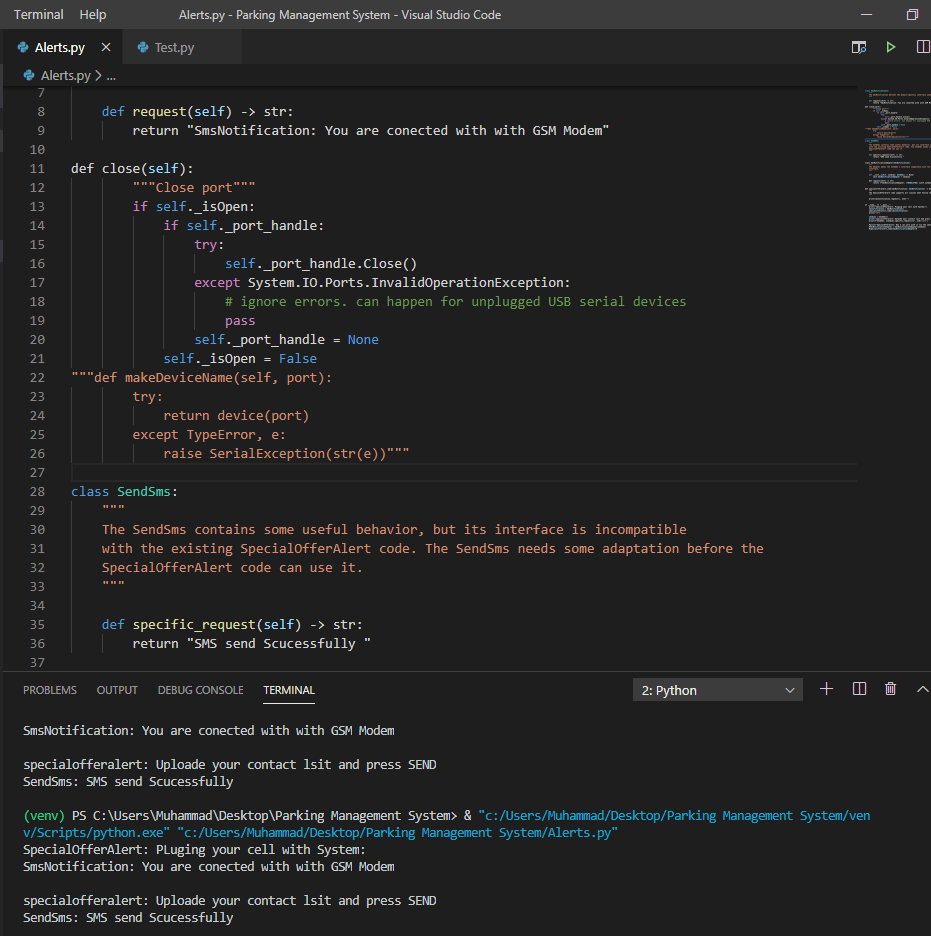
* **Figure-Ground**: Allows us to create a good contrast of the colors for the foreground a background.
* **Similarity**: It states that similar objects should be placed together so that it will be easy for the user to understand their impact and importance.
* **Proximity**: It describes that user interface objects should be placed together so that the perceived to be in the group.
* **Common Fate**: This law states that elements should be moved in the same direction so that they could be relatable.
* **Closure:** This law states that when he/she should a first look on the user interface then he/ she should understand the purpose of the element placed in user interface.
* **Focal Point**: The element with the point of the interest should be highlighted.
* **Symmetry**: The symmetry of the UI elements should be consistent.
* **A class diagram for your code (also clearly indicating where design patterns, data storage etc. is implemented) (1page A4)  
    
  I attached pictures of my work**
* **A demonstration output for the three use-cases that you have planned, i.e. program output when running a particular use case (1 page)**

1. **Get to know all employee salaries**



1. **Booking Notification**



1. **Mobile conectivity with laptop for SMS sending**

* **A short personal summary of your work focusing on difficulties and improvements for the project (0.5 pages A4)**

The work was interesting to do and there were great learning opportunities in the project. First of all it was difficult to understand and interpret the scenario of the project presented in term project file. Identifying the entities of the project and make a complete class diagram with the suitable design patterns was the difficult task. I took a lot of time to understand the system, identifying the entities and then making the class diagram with suitable design patterns. Implementing the structure of the class diagram in Python and implementing the design patterns in Python was difficult initially. As the time passed and the understanding of design patterns and Python was developed the work began to improve. The improvements made in the project were to adapt the scenario to make it complete for the implementation. By defining the roles of user and increasing the methods in the class diagram was the improvement made in the project. To focus the difficulties and improve the work, the only solution was to read the scenario of the project again and again and spend time to understand it. So this is what I did to build the clear understanding of the project. After creating the class diagram, I moved to the implementation of the project. In implementation phase, I focused to implement the class diagram structure correctly in the code. In the beginning it was difficult to implement it in the correct way but after spending a handsome amount of time I did it and implemented the design pattern in the project.