Second Report

Software Engineering

Group 2

Mehul Vora, Sajeel Ahmad, Chris Steinert,

Omar Ouf, Alex Sanchez, Kerry Liu, Prit Modi

https://github.com/Rutgers-SE/GalaxyGarage

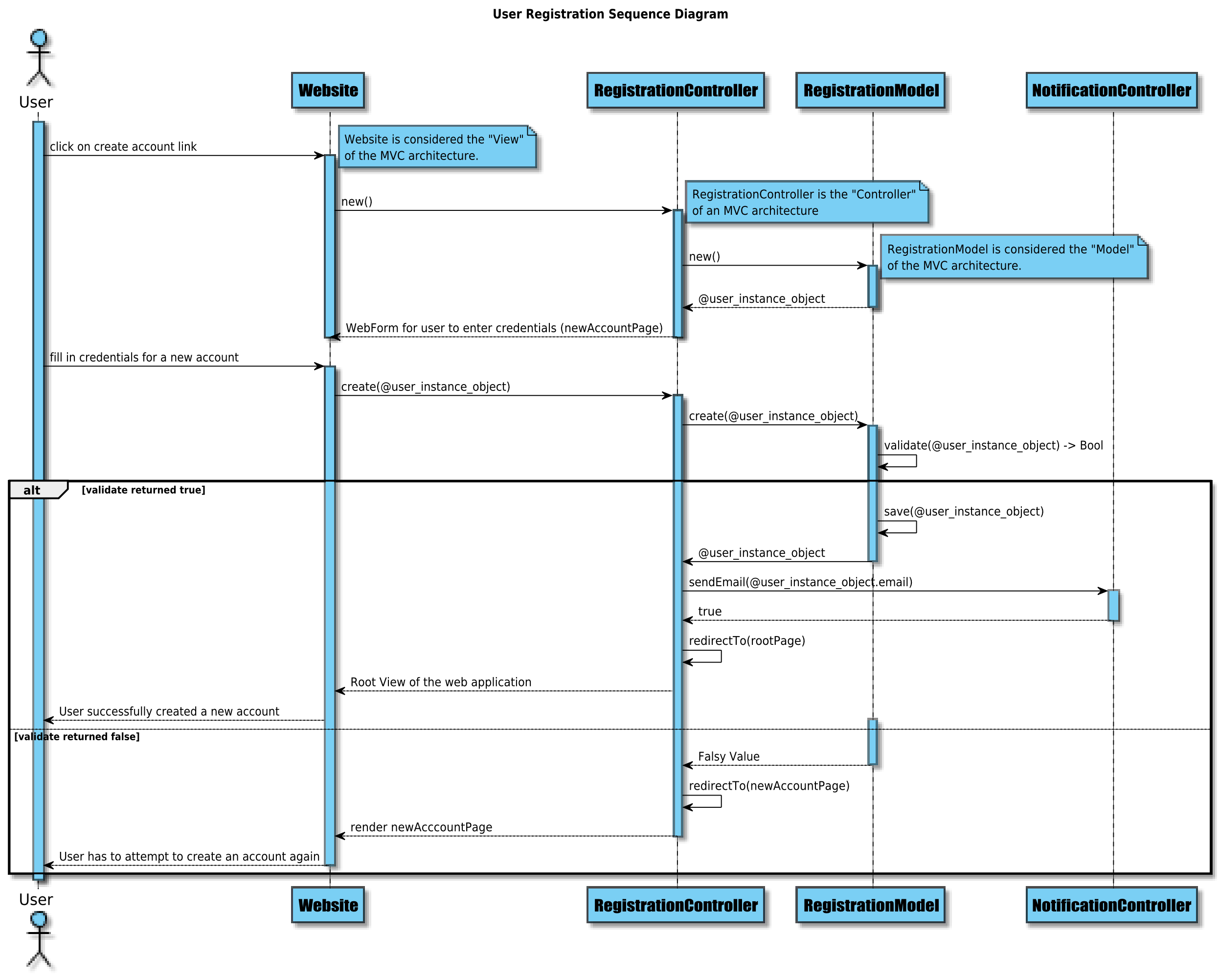
March 13, 2016

Contents

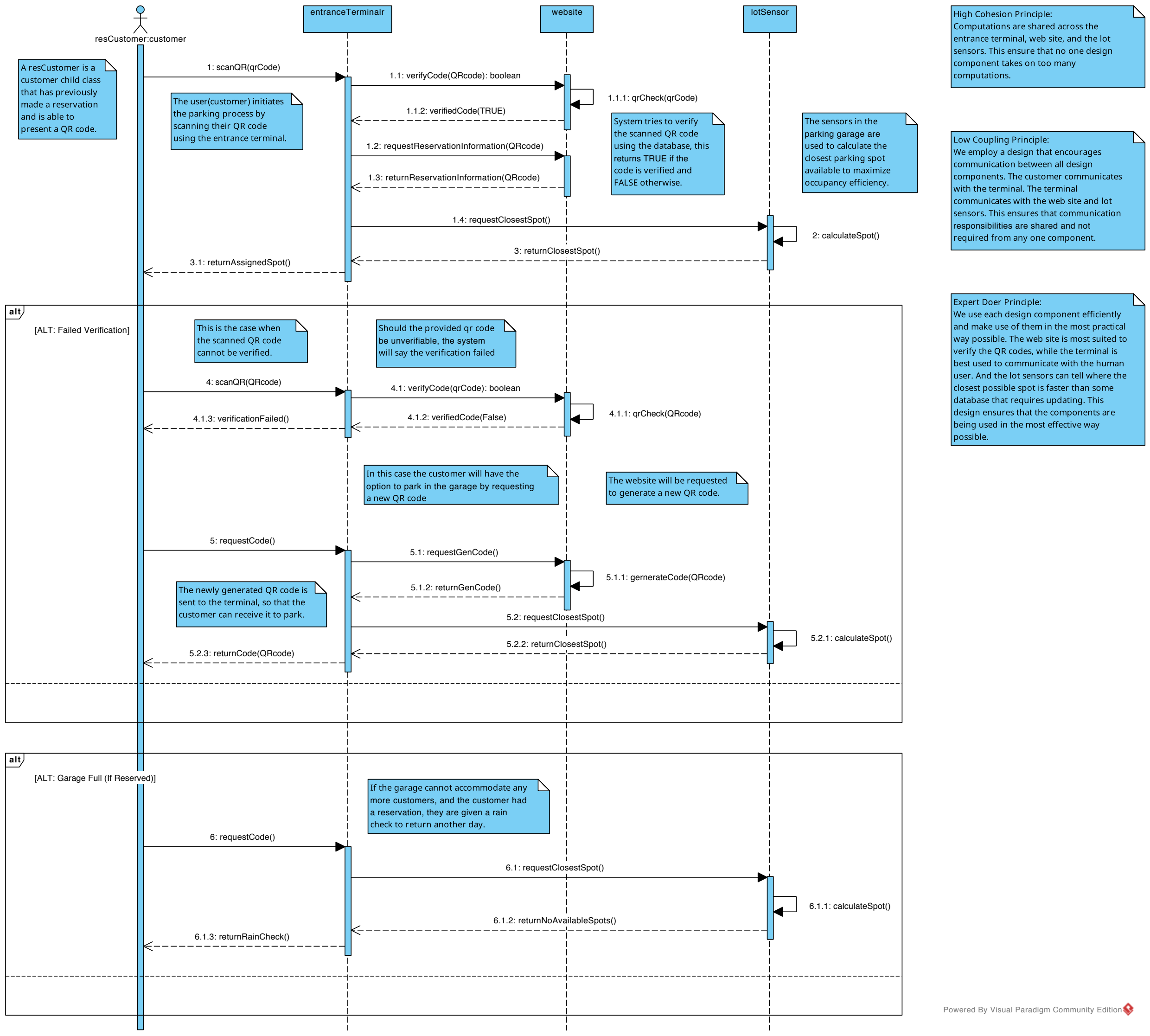
1 Interaction Diagrams

1.1 Diagrams

1.1.1 User Registration


User Registration Sequence Diagram
User
Website
RegistrationController
RegistrationModel
NotificationController
click on create account link
Website is considered the "View" of the MVC architecture.
new()
RegistrationController is the "Controller" of an MVC architecture
new()
RegistrationModel is considered the "Model" of the MVC architecture.
@user_instance_object
WebForm for user to enter credentials (newAccountPage)
fill in credentials for a new account
create(@user_instance_object)
create(@user_instance_object)
validate(@user_instance_object) -> Bool
alt
[validate returned true]
save(@user_instance_object)
@user_instance_object
sendEmail(@user_instance_object.email)
true
redirectTo(rootPage)
Root View of the web application
User successfully created a new account
[validate returned false]
Falsy Value
redirectTo(newAccountPage)
render newAcccountPage
User has to attempt to create an account again

1.1.2 Parking for Reserved


resCustomer:customer
entranceTerminalr
website
lotSensor
1: scanQR(qrCode)
1.1: verifyCode(QRcode): boolean
1.1.1: qrCheck(qrCode)
1.1.2: verifiedCode(TRUE)
1.2: requestReservationInformation(QRcode)
1.3: returnReservationInformation(QRcode)
1.4: requestClosestSpot()
2: calculateSpot()
3: returnClosestSpot()
3.1: returnAssignedSpot()
alt
[ALT: Failed Verification]
4: scanQR(QRcode)
4.1: verifyCode(qrCode): boolean
4.1.1: qrCheck(QRcode)
4.1.2: verifiedCode(False)
4.1.3: verificationFailed()
5: requestCode()
5.1: requestGenCode()
5.1.1: gernerateCode(QRcode)
5.1.2: returnGenCode()
5.2: requestClosestSpot()
5.2.1: calculateSpot()
5.2.2: returnClosestSpot()
5.2.3: returnCode(QRcode)
alt
[ALT: Garage Full (If Reserved)]
6: requestCode()
6.1: requestClosestSpot()
6.1.1: calculateSpot()
6.1.2: returnNoAvailableSpots()
6.1.3: returnRainCheck()
Powered By Visual Paradigm Community Edition

A resCustomer is a customer child class that has previously made a reservation and is able to present a QR code.

The user(customer) initiates the parking process by scanning their QR code using the entrance terminal.

System tries to verify the scanned QR code using the database, this returns TRUE if the code is verified and FALSE otherwise.

The sensors in the parking garage are used to calculate the closest parking spot available to maximize occupancy efficiency.

High Cohesion Principle:
Computations are shared across the entrance terminal, web site, and the lot sensors. This ensure that no one design component takes on too many computations.

Low Coupling Principle:
We employ a design that encourages communication between all design components. The customer communicates with the terminal. The terminal communicates with the web site and lot sensors. This ensures that communication responsibilities are shared and not required from any one component.

Expert Doer Principle:
We use each design component efficiently and make use of them in the most practical way possible. The web site is most suited to verify the QR codes, while the terminal is best used to communicate with the human user. And the lot sensors can tell where the closest possible spot is faster than some database that requires updating. This design ensures that the components are being used in the most effective way possible.

This is the case when the scanned QR code cannot be verified.

Should the provided qr code be unverifiable, the system will say the verification failed

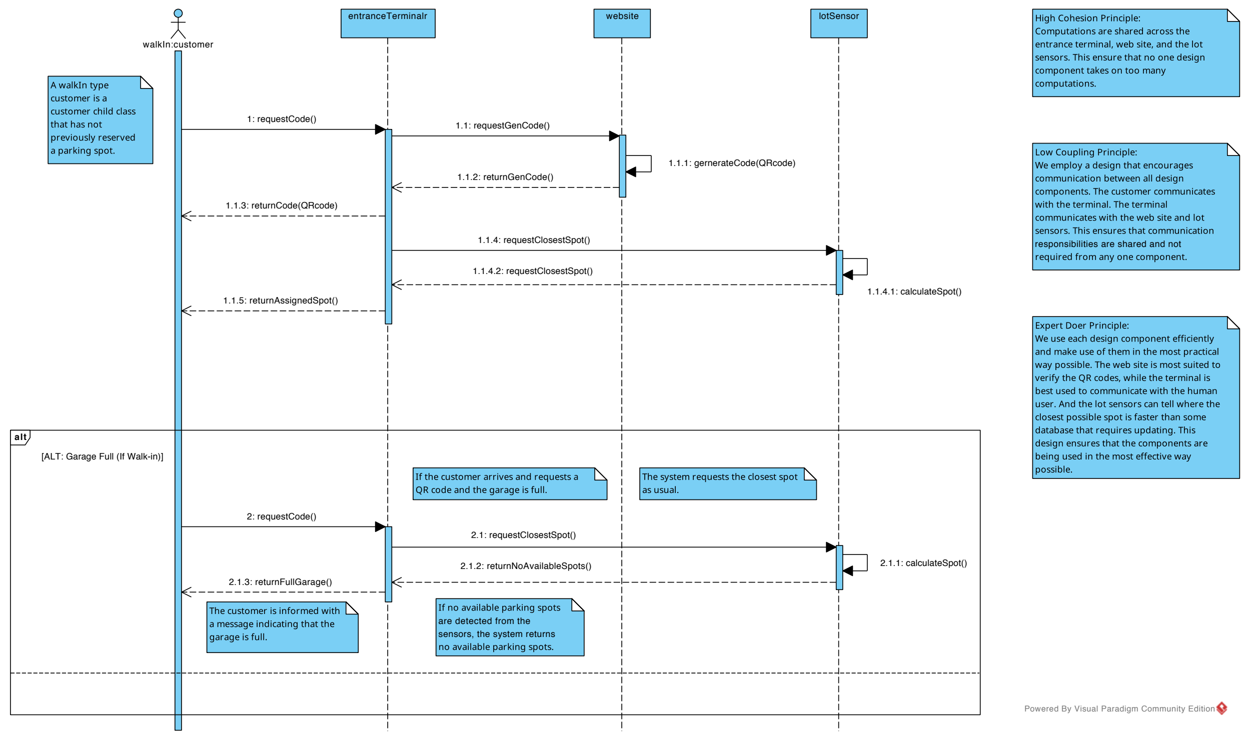
In this case the customer will have the option to park in the garage by requesting a new QR code

The website will be requested to generate a new QR code.

The newly generated QR code is sent to the terminal, so that the customer can receive it to park.

If the garage cannot accommodate any more customers, and the customer had a reservation, they are given a rain check to return another day.

1.1.3 Parking For Walk-In


walkIn:customer
entranceTerminalr
website
lotSensor
1: requestCode()
1.1: requestGenCode()
1.1.1: gernerateCode(QRcode)
1.1.2: returnGenCode()
1.1.3: returnCode(QRcode)
1.1.4: requestClosestSpot()
1.1.4.2: requestClosestSpot()
1.1.4.1: calculateSpot()
1.1.5: returnAssignedSpot()
alt
[ALT: Garage Full (If Walk-in)]
2: requestCode()
2.1: requestClosestSpot()
2.1.1: calculateSpot()
2.1.2: returnNoAvailableSpots()
2.1.3: returnFullGarage()
Powered By Visual Paradigm Community Edition

A walkIn type customer is a customer child class that has not previously reserved a parking spot.

If the customer arrives and requests a QR code and the garage is full.

The system requests the closest spot as usual.

The customer is informed with a message indicating that the garage is full.

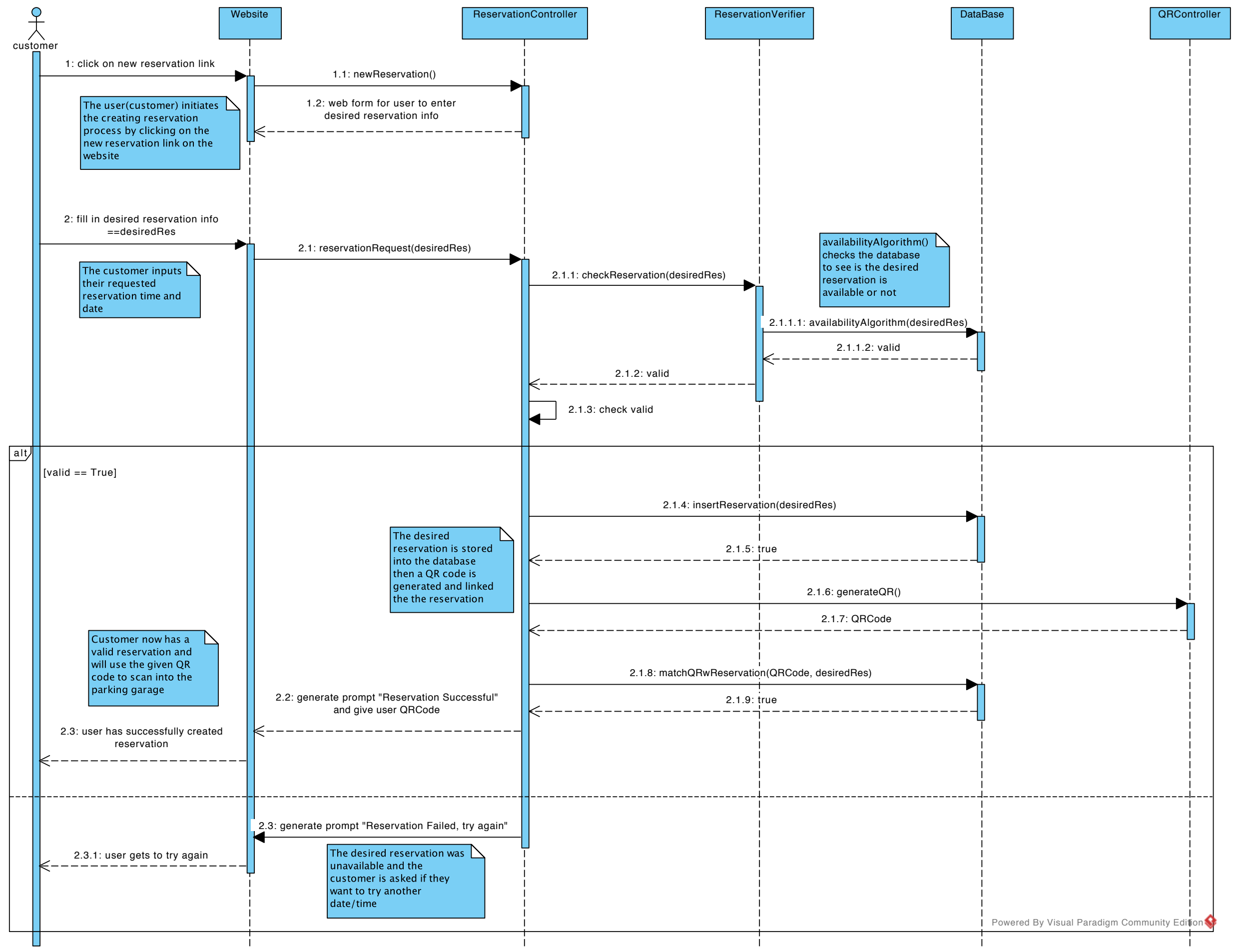
If no available parking spots are detected from the sensors, the system returns no available parking spots.

High Cohesion Principle:
Computations are shared across the entrance terminal, web site, and the lot sensors. This ensure that no one design component takes on too many computations.

Low Coupling Principle:
We employ a design that encourages communication between all design components. The customer communicates with the terminal. The terminal communicates with the web site and lot sensors. This ensures that communication responsibilities are shared and not required from any one component.

Expert Doer Principle:
We use each design component efficiently and make use of them in the most practical way possible. The web site is most suited to verify the QR codes, while the terminal is best used to communicate with the human user. And the lot sensors can tell where the closest possible spot is faster than some database that requires updating. This design ensures that the components are being used in the most effective way possible.

1.1.4 Reservation


customer
Website
ReservationController
ReservationVerifier
DataBase
QRController
1: click on new reservation link
1.1: newReservation()
1.2: web form for user to enter desired reservation info
The user(customer) initiates the creating reservation process by clicking on the new reservation link on the website
2: fill in desired reservation info ==desiredRes
2.1: reservationRequest(desiredRes)
The customer inputs their requested reservation time and date
2.1.1: checkReservation(desiredRes)
availabilityAlgorithm() checks the database to see is the desired reservation is available or not
2.1.1.1: availabilityAlgorithm(desiredRes)
2.1.1.2: valid
2.1.2: valid
2.1.3: check valid
alt
[valid == True]
2.1.4: insertReservation(desiredRes)
The desired reservation is stored into the database then a QR code is generated and linked the the reservation
2.1.5: true
2.1.6: generateQR()
2.1.7: QRCode
Customer now has a valid reservation and will use the given QR code to scan into the parking garage
2.1.8: matchQRwReservation(QRCode, desiredRes)
2.1.9: true
2.2: generate prompt "Reservation Successful" and give user QRCode
2.3: user has successfully created reservation
2.3: generate prompt "Reservation Failed, try again"
2.3.1: user gets to try again
The desired reservation was unavailable and the customer is asked if they want to try another date/time
Powered By Visual Paradigm Community Edition

1.2 Design Principles

High Cohesion Principle:

Responsibilities are shared across the objects of every sequence diagram. This will ensure that no one design component takes on too many computations and/or assignments. Moreover, this will increase system maintainability.

Low Coupling Principle:

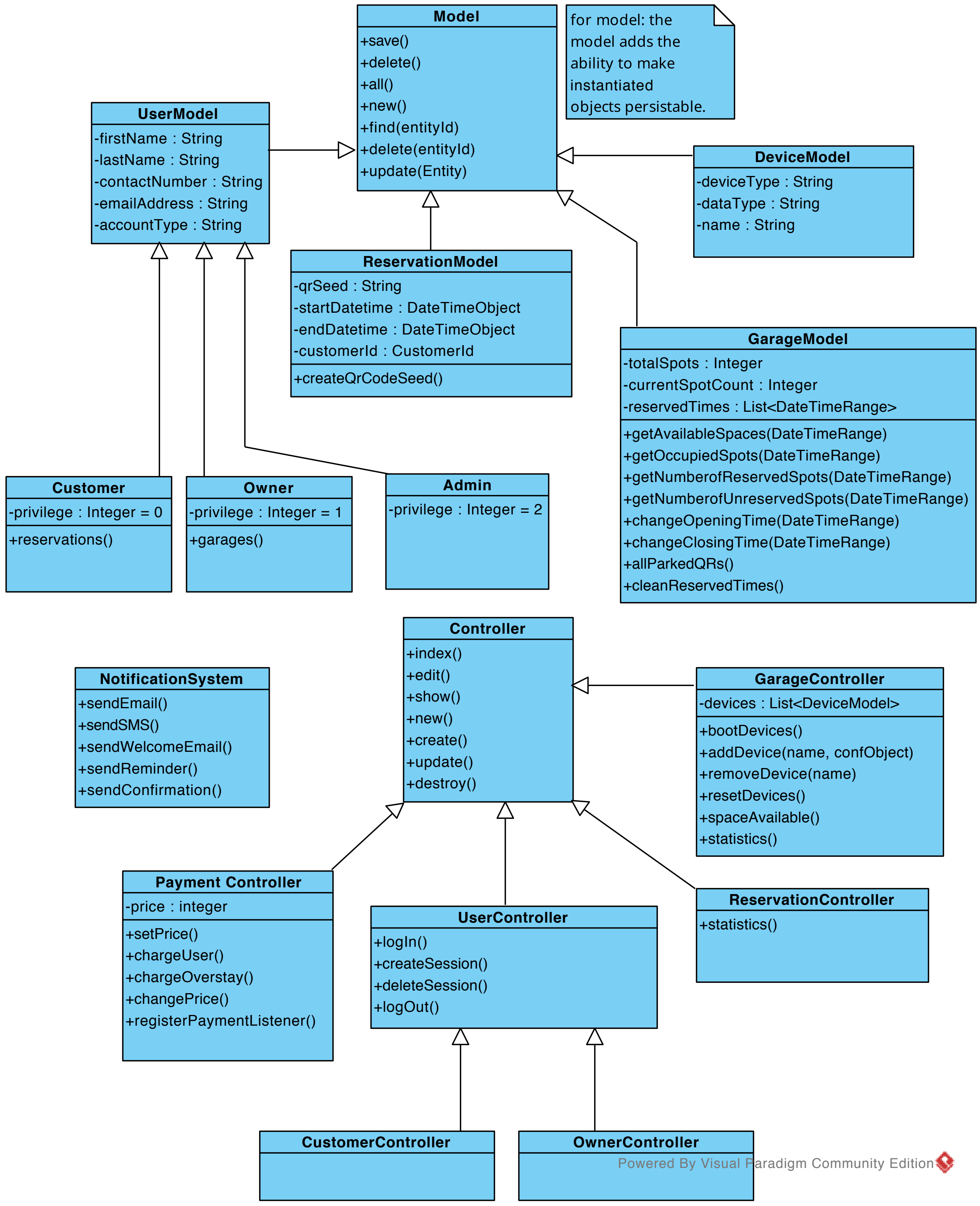
We employ a design that encourages communication between all design components. For example, the customer communicates with the website and terminal, the website communicates with the user, terminal, and database, and the terminal communicates with the website and parking lot sensors. This ensures that communication responsibilities are shared and effective. In addition, this produces lower dependency between classes, and lower impact if there happens to be a change in another class

Expert Doer Principle:

We use each design component efficiently and make use of them in the most practical way possible. For example, the website is most suited to verify the QR codes, while the terminal is best for communicating with the human user. Additionally, the lot sensors can tell where the closest possible spot is faster than some database that requires updating. This design ensures that the components are being used in the most effective way possible.

2 Class Diagram and Interface Specification

2.1 Class Diagram



2.2 Data Types and Operation Signatures

Controller

Designed around the RESTful design pattern, this class is used as the base template for creating resources.

Attributes:

Methods:

+ index() - renders an HTML page with a list of a resource.
+ edit() - renders an HTML page with a form to edit a resource.
+ show() - renders an HTML page with more detail on the the selected resource.
+ new() - renders an HTML page with a form to create a new resource.
+ create() - calls index or new depending on a successful resource creation.
+ update() - calls edit or show depending on a successful resource update.
+ destroy() - handles removing a resource and calls index.

Model

The M in MVC. This class is defined for a better control of business logic and entities.

Attributes:

Methods:

+ save() - saves the entities current state to the database.
+ delete() - deletes the current entity, and removes from the database.
+ all() - returns a list of the entities queried from the database.
+ new() - returns a new instance of an entity.
+ find(entityId) - returns the entity if it exists, null otherwise.
+ delete(entityId) - deletes the entity if it exists
+ update(Entity) - updates the current entities database record, and does something

User : Model

User controller is a subclass of controller, thus it provides the basic endpoints for creating new user entities, and creating sessions.

Attributes:

– firstName : String

– LastName : String

– ContactNumber : String - to receive notifications

– EmailAddress : String - to receive notifications

– accountType: String - to specify which account type the user is (Customer, Admin, or Owner)

Methods:

UserController : Controller

User controller is a subclass of controller, thus it provides the basic endpoints for creating new user entities, and creating sessions.

Attributes:

Methods:

+ log_in() - renders an HTML page that allows user to enter their credentials.
+ create_session() - responds to the log_in function. Checks if entity is stored in the database, if so then creates an app session for the entity (User).
+ delete_session() - called by log_out(), this function removes the current entity from the app session.
+ log_out() - redirects to web application root

Customer : User

Customer is a subclass of User. It provides the proper privilege to create new reservation.

Attributes:

− privilege : Integer = 0

Methods:

+ reservations() - returns a list of reservations that the customer has made, and that are pending

CustomerController : UserController

Customer is a subclass of User. It provides the proper privilege to create new reservation.

Attributes:

Methods:

Owner : User

Owner is an extension of the user class. It provides the means of modifying price settings, among other special functions.

Attributes:

− privilege : Integer = 1

Methods:

+ garages() - returns a list of garages owned by the owner.

OwnerController : UserController

Renders HTML, and serves to the browser. Provides the ability to log in and out.

Attributes:

Methods:

Admin : User

The Admin is an extension of the user class. Moreover, the Admin will have an elevated privilege allowing him/her to access more of the website.

Attributes:

– privilege : Integer = 2

Methods:

AdminContoller : UserController

Attributes:

Methods:

Reservation : Model

Provides an api for creating reservations.

Attributes:

– qrSeed : String - a unique tag to generate a QR Code.

– startDatetime : DateTimeObject

– endDatetime : DateTimeObject

– customerId: CustomerId

Methods:

+ createQrCodeSeed() - creates the string that will act as data to generate a QR Code.

+ validReservationSlot() - returns true if the current instantiated object is valid.

ReservationController : Controller

Renders HTML with pages that are relevant when manipulating the reservation entity.

Attributes:

Methods:

+ statistics() - shows detailed information based on data collected.

GarageModel : Model

The class GarageModel can return the number of vacant, occupied, and reserved spots for a given time. To gather this information, it will scan the database for the appropriate date and record the proper information.

Attributes:

– totalSpots : Integer - Total number of spots available for parking.

– currentSpotCount : Integer - The number of physical spots currently not taken.

– reservedTimes: List<DateTimeRange>

Methods:

+ getAvailableSpots(DateTimeRange) - returns an integer of the number of empty spots for a given date and time.

+ getOccupiedSpots(DateTimeRange) - returns an integer of number of occupied spots.

+ getNumberofReservedSpots(DateTimeRange) - returns integer of reserved spots

+ get_number_of_unreserved_spots(DateTimeRange) - returns integer of unreserved spots.

+ changeOpeningTime(DateTime) - modify the garage operating time. Return type will be double, 1-24, which will be using military time.

+ changeClosingTime(DateTime) - modify the garage closing time. Return type will be double, again based off military time

+ allParkedQrs() - returns a list of all the qr codes that entered and have not left the garage

+ cleanReservedTimes() - removes the DateTimeRages that have expired.

GarageController : Controller

Attributes:

− devices : List<DeviceModel>

Methods:

+ getAvailableSpots(DateTimeRange) - returns an integer of the number of empty spots for a given date and time.

+ bootDevices() - boot devices that are in the garage

+ addDevice(name, confObject) - add another device to the running system.

+ removeDevice(name) - remove device from the system.

+ resetDevices() - restart the devices in the system.

+ spaceAvailable() - the HTTP endpoint for the reservation system to communicate with the garage. This returns a json response containing a boolean and an integer.

+ statistics() - shows detailed information based on data collected from the devices.

Payment Controller

Allows the owner to set the price of parking. This class also handles charging the customers for their stay in the garage.

Attributes:

− price: Integer - price for parking will be stored in an integer.

Methods:

+ setPrice() - sets the price of parking.

+ registerPaymentListenter(Object, method) - triggers a payment when the the object method was called.

+ chargeUser() - charges the user based off their length of stay and price of parking.

+ chargeOverstay() - charges the user an additional fee if they overstay their reservation period.

+ changePrice() - function used to change all prices setting available. Return type will be double, which will hold the price

+ viewTotalProfit() - Does the calculations necessary for calculating the total profit depending on multiple factors such as number of customers and total expenses. Returns double which will be the total profit made.

+ BankProfit() - Will transfer total profit made to linked bank account or payment card.

DeviceModel: Model

Attributes:

− DeviceType : string - Type of device. (sensor, camera, etc.)

− SignalType : string - Data type that this device responds to.

− Name - The designation of the specific device.

Methods:

+ on(event:string ,fn) : Void - Calls the callback function when an event is emitted from the event server singleton instance.

+ emit(event-name, args) : Void - emits an event to the global event emitter.

+ init() : Void - start/initializes the device.

+ terminate() : Void - stop/terminates the device.

+ emit(event-name, args) - runs all functions that are connected to the event-name.

+ log(any type) - stores the data into a persistent database.

+ viewDeviceLog() - returns a list of events that were logged by a device.

NotificationController

Sends an email or SMS message to remind customer of reservation they made with parking garage

Attributes:

Methods:

+ sendEmail(customerId) - sends email reminder to user specified in the parameter.

+ sendSMS() - sends a SMS reminder about future reservation.

+ sendWelcomeEmail() - sends an email that describes the parking process to the customer.

+ sendReminder() - sends an email or text message reminding customer about reservation a few days ahead.

+ sendConformation() - sends an email or text message confirming reservation immediately after reservation was made.

2.3 Traceability Matrix

User Model:

- User: User is the base class made for all user classes
  - Customer: The Customer class is a subclass of the User base class that is implemented to allow customer objects to interact with the system.
  - Owner: The Owner class is also a subclass of the User base class that is implemented to allow the owner to interact with the system.
  - Admin: The Admin class is another subclass of the User base class that is implemented to have control of the website and devices used within the garage such as sensors .

User Controller

- Website: The website is the main User Interface, where the user will have control over implemented methods specified for him/her depending on his or her account type.

Notification Controller (Mail/SMS Controller

- Reminder: Both email and SMS reminders can be implemented within this single class. Moreover, this class will essentially send out email and/or SMS messages reminding the customer of their upcoming reservation.

Reservation Model

- Reservation: The reservation model will be implemented within the reservation class. The name remains the same because it is crucial to the successful operation of the garage.

Payment Controller

- This concept was introduced to better manage the payment process.

- Payment Service: The object that manages payment. To use this object you must register the object you want to watch, and when the registered object calls the watched method It will trigger the method.
- Price: The price variable is the amount of money that the customer will have to pay to obtain a parking space. This variable will be configurable within the price class. This amount is dependent on whether the customer chooses to reserve a parking or if he/she wants to walk in without a reservation and to park in an open spot.

Reservation Controller

- The reservation controller will work as an application programming interface and communicate with the internal controller to make reservations, delete reservations, and adjust reservations.

Device Model

- QR Scanner: The QR scanner is a device essentially that scans the QR codes of incoming customers and shares them with the database.
- Sensor: The sensor is a device essentially that determines whether a spot is available or not. In the larger design, these devices will share information with the database crucial to the operation of the garage.
- Camera: The camera is a device that reads the license plates of incoming cars and shares that information with the database. Moreover, it will be used for security features as well such as surveillance.
- Entrance Gate: The entrance gate is a device essentially that processes an entry of an incoming customer.

Garage Model

- Garage Model: The name remained the same for the garage model because it shows the way the garage is represented within the system and keeps track of which spots are available for new reservations. This is an essential part of this design.

Garage Controller

- The internal controller concept is implemented through interaction with devices within the garage. This will work as an API that allows the controller to communicate with the reservation controller to operate internal tasks within the garage model. The controller receives messages from the devices within the garage and controls the internal interactions between objects.

3 System Architecture and System Design

3.1 Architectural Styles

- Service-Oriented Architecture (SOA) - This is how the internal interaction system will communicate with the reservation system. The reservation system will define a very simple API exposed in a RESTful way.
- Message Bus - This is how the internal interactor controller will interact with the sensors within the system.
- Component-Based Architecture
- Object-Oriented - This is how new devices will be added to the system.

3.2 Identifying Subsystems

- Payment
- Device Registration

- Device Orchestration

3.3 Mapping Subsystems to Hardware

- For the payment system, we will be using stripe and encapsulating that api in the PaymentController.
- For adding sensors to the system, we will be creating a generic io device class which will have a set of unique operating types. To augment the functionality a bit we will be overriding some of the default methods with certain functions that can interface with the physical hardware.

3.4 Persistent Data Storage

- For internal interaction system (GarageController, GarageModel), we will be using Mongodb. It provides more flexibility when dealing with unique data.
- For the Reservation System and User creation system, we will be using Sqlite3. It is by far the one of the more simpler databases to set up. If there ever comes a time where we need to switch databases, we will be using migration tools, and ORM libraries to keep a constant code base for communicating with the database.

3.5 Network Protocol

- HTTP - This will be used for creating the page based web application. This will include User Registration, Reservation Creation, and Owner Management.
- Sockets - This protocol will be used when we need a more real time network transfer protocol. Specifically we will be using this protocol when reading input from sensors.

3.6 Global Control Flow

- Our system is partially event-driven, and heavily relies on time. For our system to stay performant, we will run our system over multiple threads.

3.7 Hardware Requirements

- HTTP Network - For the online Registration and Reservation System to work
- Local Network - For the sensors to be able to communicate with the internal interaction portion of the parking system.
  - Must have a network DMZ for incoming and outgoing requests.
- Screen Display (Side of Garage) - Display the current capacity of the parking garage.
  - Must at least have 960x640 pixels
- Disk Storage - Act as a location for user information will be stored.
  - Must be at least 2 TB
  - raid 10 configuration
  - 7200 RPM
- Camera - Needed to associate the walking/reserver to the vehicle that he/she is driving. Also used for security purposes. It must have:
  - 16.9 million pixels.
  - 35 nm focal length.
  - aperture f/n where, n ranges from 16 and 1.4
- Space Sensor - Needed to determine what parking spaces are available or taken. This will heavily influence what spaces are generated for the next reserve/walkin.
  - The difference threshold must be around 1Ft, and 2Ft.

  - The maximum distance that the sensor must be able to detect must be at least 14 Feet.
  - The minimum of 2.5Ft.

4 Algorithms and Data Structures

4.1 Algorithms

4.1.1 Reservation Logic

- Customer attempts reservation time
- System checks reservation time with projected spot availability/currently reserved times.
- System registers reservation time if available or asks user to choose a different time.
- Reservations are now perfectly structured.

4.2 Data Structures

This system will be implemented using models that will essentially be data structures that allow the different objects in the system the function in a more organized way. Models include UserModel, ReservationModel, DeviceModel, and GarageModel. Controllers will be used to interact with the different models in the system. There will be a Model base class introduced that outlines the basic form of the individual models. The base model class will have certain methods that all child class models will need, such as save, delete, new, etc. This model adds the ability to make instantiated objects persistable. Each model is derived from the base model and has specific methods and attributes that are needed for specific tasks within the system. The UserModel will be have three child classes: Customer, Owner, and Admin. Their main difference will be their privilege attribute that will essentially decide what that user is allowed to alter and what they are not allowed to alter.

The DeviceModel is designed to essentially be able to interface with any kind of device in the system, such as camera, sensor, gate, etc. This allows the software to be incredibly future proof and reusable. A camera device, for example, would be implemented as a subclass of DeviceModel and given specific methods that the camera would need in order for the garage to operate, such as read license plate. The other devices would be implemented the same way.

Similar to the model class there will be a base controller class that will be responsible for controlling specific models within the system. The base controller class will have certain methods that all controllers in the system will use, such as index, edit, show, etc. The controllers that will be implemented include GarageController, ReservationController, UserController, and PaymentController. These controllers will essentially allow us to read and modify the model instances in the system. Also there will be a NotificationSystem that will handle reminders and further communications with the customers. This implementation makes the flow of concept to code far more smooth and simple. It also makes the software future proof.

The UserController will have two subclasses: one for customers, the CustomerController, and one for owners, the OwnerController. This Controller will allow the users to log in, log out, create sessions and delete sessions. Our data will be managed using two databases - SQL & MongoDB. SQL will be used when the data is structured, whereas MongoDB will be used when the data comes in multiple formats with small variances. User data and reservations will go into the SQL database since there is little variance between entries. Mongo will be used for the array of sensory data from the garage due to the large variances in data entries.

5 User Design and Implementation

We will be following a MVVM architecture for the frontend and a MVC architecture for the backend.

5.1 Web Interface

- Home.index

  This page will be the greeting screen for every customer arriving to the website. This page will have links for the user to login or logout, depending on the current session, and links to go through the reservation creation process. This view is accessible to anyone trying to access the website.

- Reservation.index

  The Reservation.index page has different views depending on who is viewing the content. If the Owner or Admin is viewing the content, then we will be greeted with a list of all the saved reservations in the system. If the Customer is viewing the page, they will be greeted with the reservations that only they created.

- Reservation.new

  The Reservation.new page is where the currently logged in customer will be able to enter the data to create a new reservation. Depending on what the customer set for the DateRange, the frontend will fetch the status of the garage during that DateRange, and disable/enable progression of the creation process.

- Reservation.edit

  The Reservation.edit page is where the currently logged in user will be allowed to modify previously created reservations that have not expired. This page will reuse the form from the Reservation.new view.

- subwindow:Reservation.remove

  This is a dropdown view, initialized after pressing the delete reservation button from either the Reservation.index page or the Reservation.edit page. When views this view, the customer will be greeted with an option to confirm the deletion of the reservation, or cancel.

- Customer.new

This page is where the customer will be able to enter in their information, and create an account. After submitting the form on this page, the customer will receive an email containing a confirmation link. When the confirmation link is clicked, it will send the customer to the Customer.login screen.

- Customer.login

  This page allows the customer to login using their login credentials. After a successful login, they will be sent to Home.index view.

- Admin.login

  This page allows the Admin to login using their login credentials. After a successful login, they will be sent to AdminPanel.index view.

- Owner.login

  This page allows the owner to login using their login credentials. After a successful login, they will be sent to OwnerPanel.index view.

- Financial.index

  This page is only accessible to the owner. It provides a graphs of predicted vs actual profit for the date, week, month, quarter, and year. Also this page provides Ajax forms where the owner can adjust variables exposed by each module of the system. An example would be the ability to modify the price of the reservation. This value would be exposed by the PaymentSystem.

- ObservationDeck.index

  This page is only accessible to the Admin. This view contains a list of system specific feature. From this view the admin can see device health and other device specific features.

6 Design of Tests

6.1 Testing Strategy

We will be using a simulated garage and mock data to test the system in the same way if it was to be simulated in real life. Moreover, we will be using unit testing with the following cases: Classes and their methods, Website, and the Database.

6.1.1 Testing Classes and Their Methods

- *Goal:* To ensure all classes and their respective methods work individually and properly as units.
- *Methods to test:*

  Please refer to Data Types and Operational Signatures mentioned earlier in report. Moreover, we will be testing to ensure all methods work properly and as stated using common testing methods such as printf.

6.1.2 Testing the Website

- *Goal:* To ensure users are allowed to make/modify accounts and reservations and other features such as price and operating times for admin accounts.
- *Methods to test:*
  - Create Account - Creation of account will occur upon registration. To confirm an account has been created, we will verify it by email confirmation, and further, by logging into the account.
  - Update Account - To verify updates are saved, we will modify the account information, and then log out and then log back in to check for updated information.
  - Create Reservation - To ensure a reservation has been created, we will verify it with the database. Modify Reservation - To ensure the reservation has been modified, we will verify it with the database.

  - Change Price - To ensure changing price of reservation works, we will verify it by creating a new reservation and confirming the price change, or displaying the new price on the website.
  - Change Operating times - To ensure operating times change, we will verify and confirm it by acknowledging the changed operating times depicted on the website.

6.1.3 Testing the Database

- *Goal:* To ensure that all data regarding the parking garage is organized and stored proficiently. Moreover, this includes customer data such as personal information and billing information, in addition to information collected by the garage devices like surveillance photos/videos and license plates.

  To ensure that the Database is structured well and that data can be accessed, changed, or inputted in a reasonable and timely fashion.

- *Methods to test:*
  - Record - Check to see if and how data is recorded into the database.
    - We will be testing the database by inputting various, different sets of data to see and ensure that data is being organized and stored in an efficient manner.

6.2 Integration Testing Strategy

We will be using Horizontal Integration Testing, more specifically Bottom-Up Integration Testing for our project. To do this, we will first begin testing each individual unit one at a time to ensure proper functioning. Gradually, we will then begin integrating specific classes together to create subsystems, and then testing those subsystems. Additionally, we are planning to have three main subsystems at the end result to make up the system as a whole, these three will be the garage and its inner workings, the website, and the database. Eventually, as testing progresses, we will have performed a series of integration testing until we reach a point where we are testing the system as a whole.

7 Project Management & Plan of Work

7.1 Contributions

	Alex Sanchez	Mehul Vora	Kerry Liu	Sajeel Ahmad	Chris Steinert	Omar Ouf	Prit Modi
Project Management		70%		30%			
Transferring Documentation to LaTeX and Table of Contents	5%	5%	90%				
Interaction Diagrams	23%	10%		24%	23%	20%	
Interaction Diagram Design Principles		30%		70%			
Class Diagrams	40%	15%	1%	34%			10%
Data Types and Operation Signatures	25%	5%	5%	25%	20%	20%	
Traceability Matrix	10%		10%	80%			
Architectural Styles	100 %						
Identifying Subsystems	100 %						
Persistent Data Structures	100 %						
Network Protocol	100 %						
Global Control Flow	100 %						
Hardware Requirements	100 %						
Algorithms			20%	80%			
Data Structures			10%	90%			
User Interface Design and Implementa	100%						
Design of Tests		90%	10%				

7.2 Team Communications

Face to Face Meetings

To discuss urgent matters and assignments/deadlines.

Slack

A messaging app for teams - Used for communications while working on project.

FaceBook

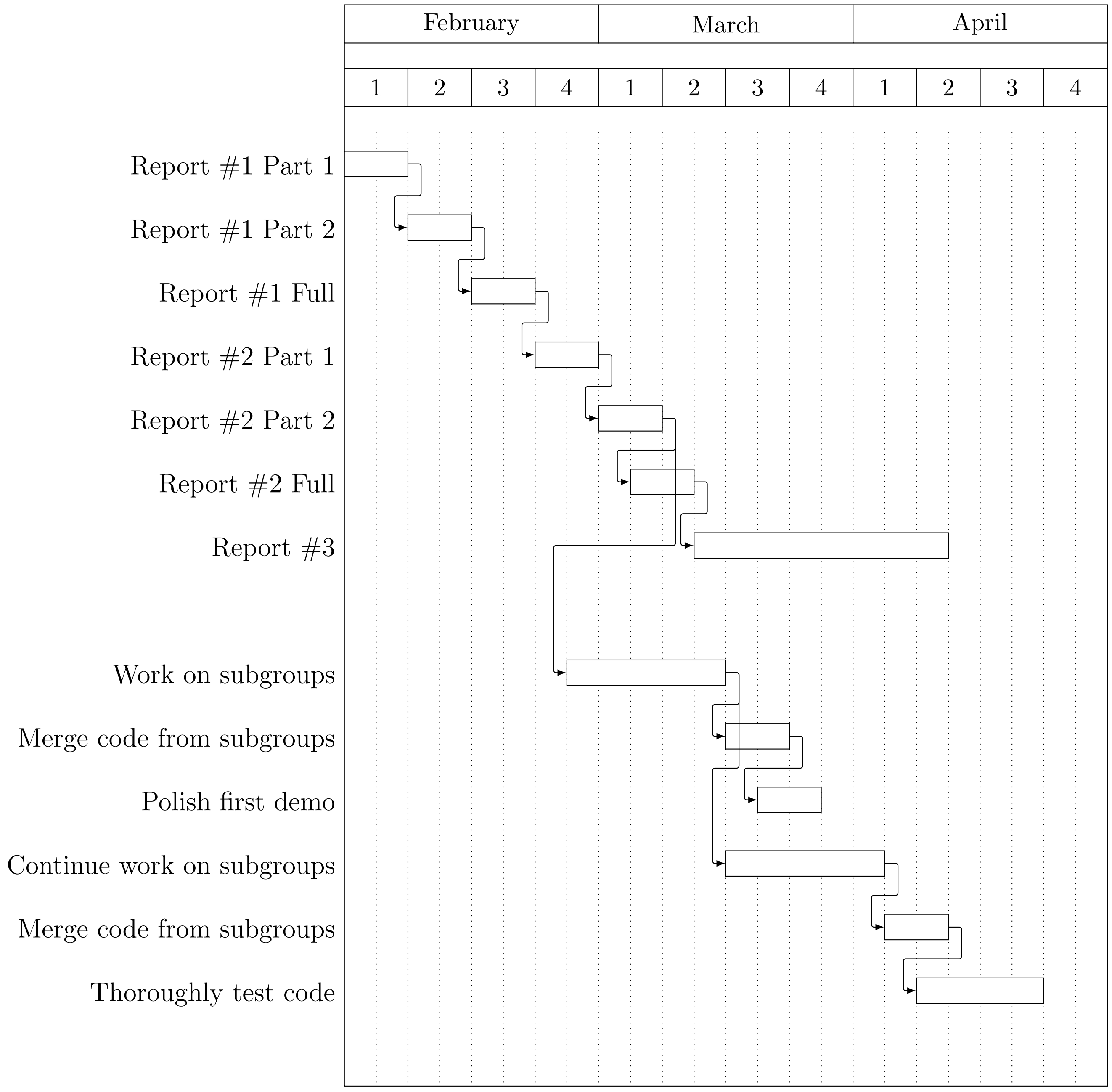
Used for communications about meeting times and times of availability.

Google Drive

Used as a team repository to store all documents and meeting/deadline information.

- Google Docs and Spreadsheets are used to distribute assignments and deadlines to all team members, along with additional information.

7.3 Plan of Work



8 References

- https://en.wikipedia.org/wiki/Gantt_chart
- https://www.sharelatex.com/learn/
- http://www.ece.rutgers.edu/~marsic/Teaching/SE/