SmartSeat

SmartSeat Bus System

Object Design Document

Ben Sewards
John Longo
John Piccolomini

Lab Section: COMP 566-02
Workstation: WENTW 210/DOBBS 308

Last Updated: (2/27/13)

TABLE OF CONTENTS

1. INTRODUCTION 2

1.1 Object design trade-offs 3

1.2 Interface documentation guidelines 4

1.3 Definitions, Acronyms, and abbreviations 5

1.4 References 5

2. PACKAGES 6

3. CLASS INTERFACES 7

3. DATABASE GUIDELINES 8

1. INTRODUCTION

1.1 Object design trade-offs

Performance

Performance is one of the lesser important criteria when designing our system. Our concern is less on query response time and more on accuracy. Keep in mind that because of the simplicity of the key data, retrieving daily information initially from the central database to update the local OBBS DB should take an acceptable amount of time of no longer than 10 seconds of wait time.

In terms of requests and responses from the web server, we will initially be using emulators to mock each system. Data will traverse through emulator instances on a LAN by going through each android activity and using JSON as a data encapsulation format to send over REST-based services to a port on another emulator instance.

Dependability

It is better that the user get slow performance than be given the appearance that the application is nonfunctioning, which leads us to dependability. All systems must have maximum uptime in order to fulfill their functions of overall sending data to the MFP. In order for errors to be recovered automatically without human intervention, data validation and the sending of data needs to be efficient. If, for example, data is lost between systems, the service must act by retransmitting that same information. If there is a critical error where a system cannot be reached, the MFP should be notified of down time.

Cost

Keeping costs low is a priority as the budget is non-existent at this time. Since the project scope is to build a prototype, there should not be any road blocks ahead that deal with cost.

Maintenance

Maintenance will be a non-intrusive factor while building the prototype. Since downtime is allowed, we will be focusing more on the dependability and performance of the system at whole in order to reach a level where maintenance is required (before a user base is achieved).

End User

The end user experience must be as streamlines as possible at the MFP and OBBS for both Driver and Parent. The idea behind the MFP application is to make sure that the user is comfortable with navigating through the primary interface after connecting to their student(s). Data will be presented as simple notification messages in list format and stored in their local DB.

Space vs. Speed

Space is notably expendable for each database. Even with that said, all data should be stored in formats that allow for the quickest possible processing regardless of size. This will take into consideration processing algorithms and SQL commands needed to retrieve data and place into existing tables.

Delivery Time vs. Functionality

In the package flow architecture thoroughly described in section 2, some packages are sub-systems of another package and rely on waiting for a response or have to wait till data is released in order to perform that systems job. Functionality will not be compromised, but in the event that all functionality is completed, anything in queue will be added.

1.2 Interface documentation guidelines

Identifiers, Methods and Class Variables

Camel Casing will be used for Identifiers and Methods
Examples: name, firstName, dataResource

Constants

Any constants, global or private, must be entirely upper case. Each word must also be separated by an underscore.
Examples: PI, DEFAULT_AGE

Classes

Pascal Casing will be used for Classes. They are nouns and must be capitalized as well with every subsequent word in the class name.
Examples: NotificationService, DatabaseConnection, Student

Exception Classes

Following the same convention as Classes except they end with the word "Exception".
Examples: ParentAlreadyExistsException

Local variables

Must be short and lower case, preferably less than 8 characters that describe the data.
Examples: i, count, temp

Methods

When possible, methods should contain a verb phrase followed by a subject. Parameters should be clear nouns.
Examples: getName(), setName(string name)

Comments

Comments should be used to document all methods that are not accessors or mutators and other random comments may use the inline comment style “//”. Examples:
/** Class description
*
* @version 1.0
*/

1.3 Definitions, Acronyms, and abbreviations

LAN | Local Area Network
JSON | JavaScript Object Notation is used for data interchange and is easy for humans to read and write.
Port | The application destination address serving as a communication endpoint.
REST | Using JSON to send and receive data
Tables | A term used when arranging data from an entity in a DB.

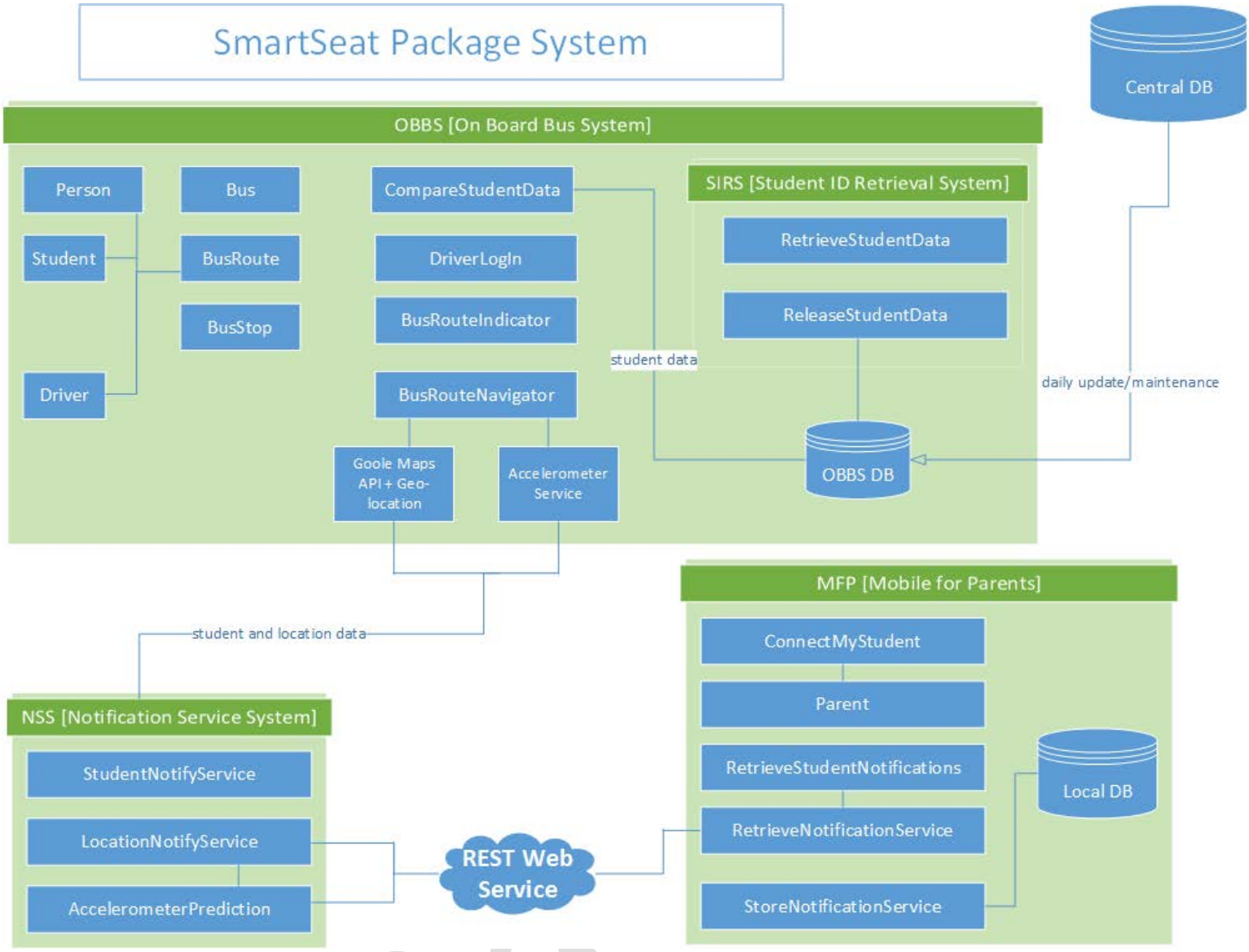
1.4 References

The Android API documentation and stackoverflow.com are heavily used resources that our team will be utilizing for education when going into the developing process.

References regarding the overall landscape for this ODD include Wikipedia and *Managing Object Design*, in *Object-Oriented Software Engineering* by Bruegge.

2. PACKAGES

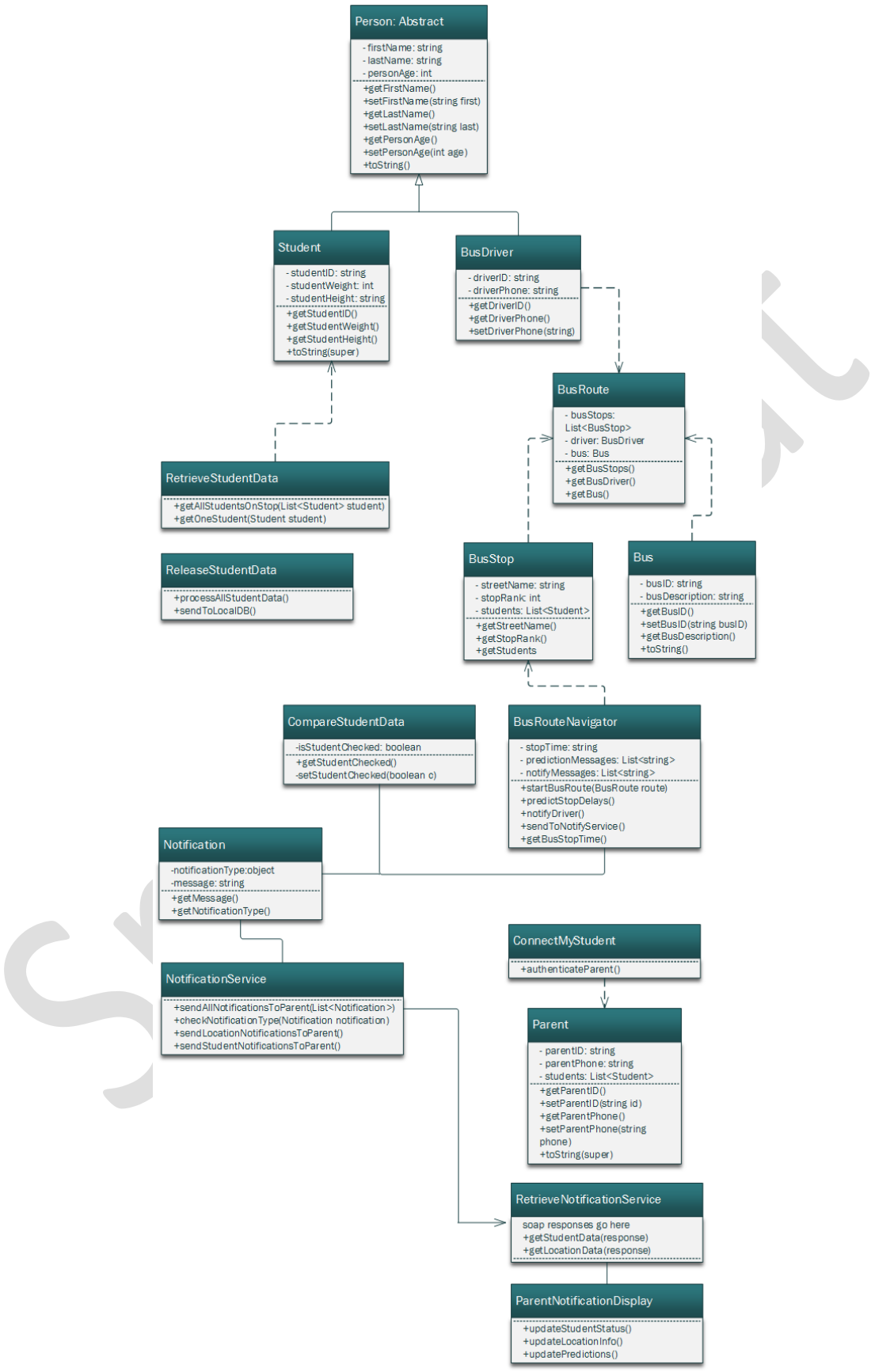
The decomposition of each system described in the SDD is displayed into packages and the file organization of each class is organized in the given package. Dependencies of other packages, including sub-systems and the package flow, are also displayed below:



3. CLASS INTERFACES

To thoroughly describe the classes and their public interfaces and user interfaces, this section includes a UML Class showing inheritance, abstraction, class dependencies, class attributes, operations, and class workflow.

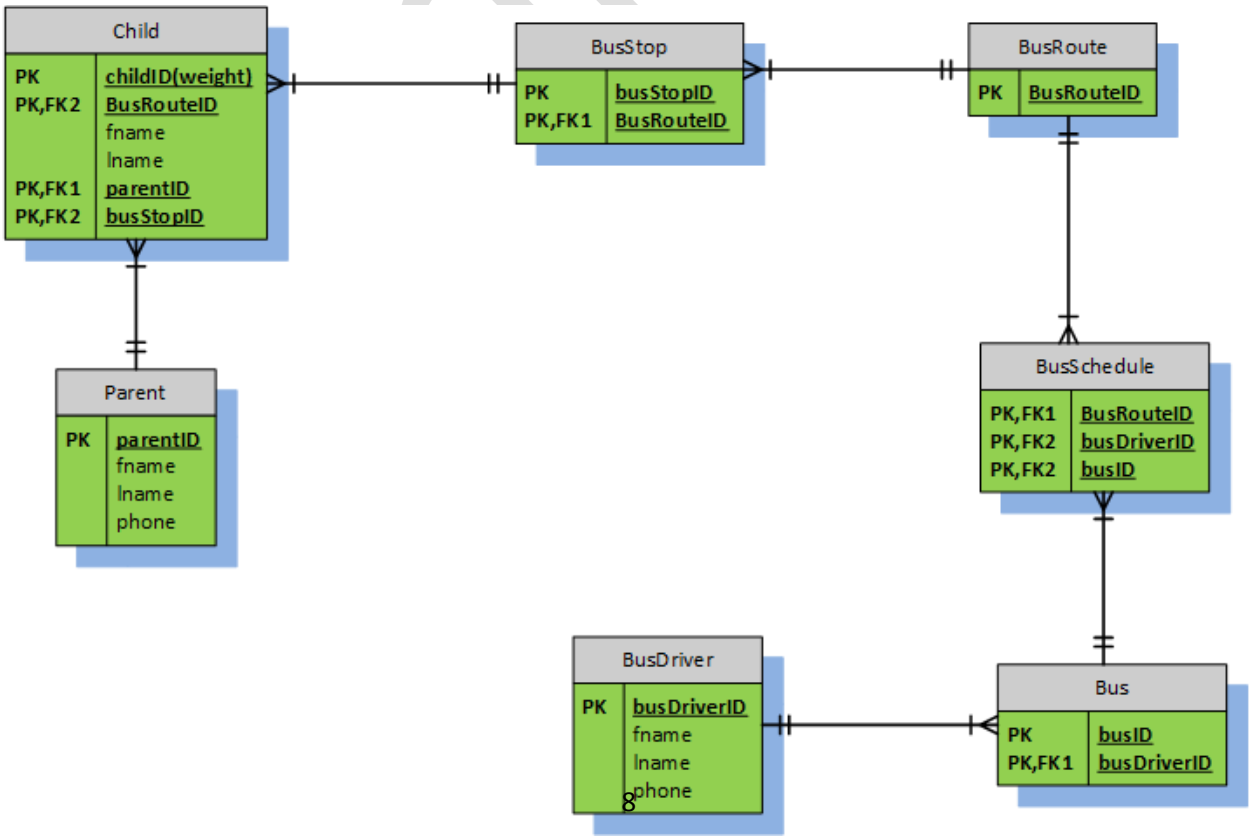
Since I wanted to make the relationships and workflow look clear, the diagram takes up a good length when at 100%. For this reason, it has been scaled down to fit on one page of the ODD below. If the ODD is in PDF format, it will be larger.

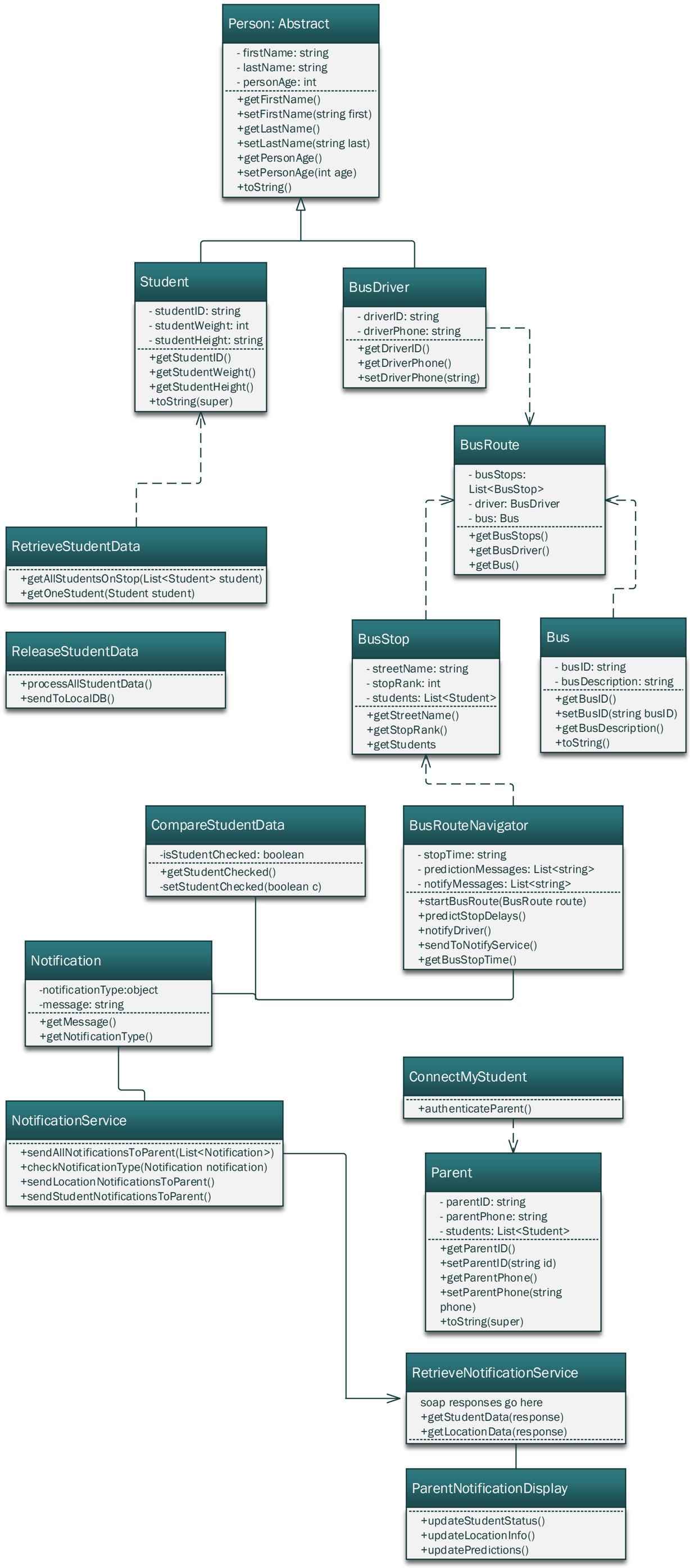

Person: Abstract
- firstName: string
- lastName: string
- personAge: int
+getFirstName()
+setFirstName(string first)
+getLastName()
+setLastName(string last)
+getPersonAge()
+setPersonAge(int age)
+toString()
Student
- studentID: string
- studentWeight: int
- studentHeight: string
+getStudentID()
+getStudentWeight()
+getStudentHeight()
+toString(super)
BusDriver
- driverID: string
- driverPhone: string
+getDriverID()
+getDriverPhone()
+setDriverPhone(string)
BusRoute
- busStops: List<BusStop>
- driver: BusDriver
- bus: Bus
+getBusStops()
+getBusDriver()
+getBus()
RetrieveStudentData
+getAllStudentsOnStop(List<Student> student)
+getOneStudent(Student student)
ReleaseStudentData
+processAllStudentData()
+sendToLocalDB()
BusStop
- streetName: string
- stopRank: int
- students: List<Student>
+getStreetName()
+getStopRank()
+getStudents
Bus
- busID: string
- busDescription: string
+getBusID()
+setBusID(string busID)
+getBusDescription()
+toString()
CompareStudentData
-isStudentChecked: boolean
+getStudentChecked()
-setStudentChecked(boolean c)
BusRouteNavigator
- stopTime: string
- predictionMessages: List<string>
- notifyMessages: List<string>
+startBusRoute(BusRoute route)
+predictStopDelays()
+notifyDriver()
+sendToNotifyService()
+getBusStopTime()
Notification
-notificationType:object
-message: string
+getMessage()
+getNotificationType()
ConnectMyStudent
+authenticateParent()
NotificationService
+sendAllNotificationsToParent(List<Notification>)
+checkNotificationType(Notification notification)
+sendLocationNotificationsToParent()
+sendStudentNotificationsToParent()
Parent
- parentID: string
- parentPhone: string
- students: List<Student>
+getParentID()
+setParentID(string id)
+getParentPhone()
+setParentPhone(string phone)
+toString(super)
RetrieveNotificationService
soap responses go here
+getStudentData(response)
+getLocationData(response)
ParentNotificationDisplay
+updateStudentStatus()
+updateLocationInfo()
+updatePredictions()

4. DATABASE GUIDELINES

For our system, we are using one central database and two local databases. The central database will house all current up-to-date information on students, busses, and parent contact information. This information will be updated and maintained by the school after initial setup is complete. As most schools typically have a database established that contains student records already, this database can either simply reference that existing database (with the creation and inclusion of all the bus and bus route information) or be a separate database completely depending on the capabilities of the school. To help with data retrieval, two local databases will be utilized. Every day, at the beginning of the day, the on board bus system database on each bus will be updated with student data in accordance with the bus route it will be servicing. This local database will act as an "attendance" list that will be referenced when students get picked up or dropped off and is designated by the SIRS (Student ID Retrieval System). When the notification system is activated, the MFP (Mobile for Parents) system will access its local database of parent contact information based on the student's identifying features.

In a perfect world, the same bus will run the same route for both pick up and drop off every day. Unfortunately, maintenance and unforeseen issues can occur. In the case of a bus being unable to run a route, the system is set up that an alternate bus will take the identity of the broken bus by using its busID. This is the only update needed because the system is designed so that the if a different bus must be used, as long as the correct busID is entered then the busRouteID will still be correct. Also, the busRouteID is dependent on the busID and not the busDriverID and therefore in the case of a bus driver having to miss work, there will be no required system updates to ensure data integrity.




Person: Abstract
- firstName: string
- lastName: string
- personAge: int
+getFirstName()
+setFirstName(string first)
+getLastName()
+setLastName(string last)
+getPersonAge()
+setPersonAge(int age)
+toString()
Student
- studentID: string
- studentWeight: int
- studentHeight: string
+getStudentID()
+getStudentWeight()
+getStudentHeight()
+toString(super)
BusDriver
- driverID: string
- driverPhone: string
+getDriverID()
+getDriverPhone()
+setDriverPhone(string)
BusRoute
- busStops: List<BusStop>
- driver: BusDriver
- bus: Bus
+getBusStops()
+getBusDriver()
+getBus()
RetrieveStudentData
+getAllStudentsOnStop(List<Student> student)
+getOneStudent(Student student)
ReleaseStudentData
+processAllStudentData()
+sendToLocalDB()
BusStop
- streetName: string
- stopRank: int
- students: List<Student>
+getStreetName()
+getStopRank()
+getStudents
Bus
- busID: string
- busDescription: string
+getBusID()
+setBusID(string busID)
+getBusDescription()
+toString()
CompareStudentData
-isStudentChecked: boolean
+getStudentChecked()
-setStudentChecked(boolean c)
BusRouteNavigator
- stopTime: string
- predictionMessages: List<string>
- notifyMessages: List<string>
+startBusRoute(BusRoute route)
+predictStopDelays()
+notifyDriver()
+sendToNotifyService()
+getBusStopTime()
Notification
-notificationType:object
-message: string
+getMessage()
+getNotificationType()
ConnectMyStudent
+authenticateParent()
NotificationService
+sendAllNotificationsToParent(List<Notification>)
+checkNotificationType(Notification notification)
+sendLocationNotificationsToParent()
+sendStudentNotificationsToParent()
Parent
- parentID: string
- parentPhone: string
- students: List<Student>
+getParentID()
+setParentID(string id)
+getParentPhone()
+setParentPhone(string phone)
+toString(super)
RetrieveNotificationService
soap responses go here
+getStudentData(response)
+getLocationData(response)
ParentNotificationDisplay
+updateStudentStatus()
+updateLocationInfo()
+updatePredictions()