BILL
Design Document

Authors:
Jamie Gross
Preston Barbare
Logan Murray

Group: 7

This page intentionally left blank.

Contents

1 Introduction

This section addresses the purpose of this document including the intended audience, an introduction to the problem and a detailed view of the project's design. In the discussion, the design of the final system including several detailed diagrams will be described in detail.

1.1 Purpose

This document outlines the design for the university billing system, BILL. One should be able to understand the high level design of the system and its participating components. It'll first cover the underlying architecture, then the associated components, and then class structure and design.

1.2 System Overview

BILL is Java API that can be provided to a multitude of systems deployed on their platform of choice. The target users are student and admin users of a university who wish to access account billing information such as viewing tuition and making payments for outstanding balances. Students, as an overview, can access their own account to view and make changes. There are different types of admins with privileges over certain students who can also view and make changes to those such students they have privileges for.

1.3 Design Objectives

The objective of this design is the following:

- Provide enough information and insight for a Java programmer to implement a functional system
- Provide enough information and insight for unit tests to be developed before all functionality is available
- Provide enough information and insight for the system to be deployed and available to client with valid access
- Provide enough information and insight for all requirements in the requirements document to be satisfied in a well organized and efficient manner

1.4 References

Use Cases and Requirements Document (request access): Link

Test Cases Document (request access): Link

1.5 Definitions, Acronyms, and Abbreviations

API (Application Programming Interface): Set of routines, protocols, and tools for building software applications (source)

Billing Interface Linkable Library (BILL)

Data Housing/Controller Subsystem (DHCS)

Authentication/Validation/Permissions Subsystem (AVPS)

Bill Management System (BMS)

2 Design Overview

*2.1 **Introduction***

After a series of trial and error that involved walking through hypothetical scenarios, we created a number of initial designs. However, after observing a pattern, we decided on a few fundamental principles that we wanted our design to follow. We will discuss 3 of those principles here.

The first principle is security first. Thus, all requests flow from the user-facing API to the Authentication/Validation/Permission Subsystem (AVPS) which ensures that access restrictions and modification restrictions are not violated. Additionally, this aids in making other subsystems more simple and lean by allowing the other subsystems to operate under the assumption that the input to those subsystems is valid, and has been checked for authentication, validation, and permissions.

Next, we placed lean design as a priority. Taking an “english prose” approach, we discussed our design in terms of creating the simplest and most modular components. Thus, every task is broken down into very simple compositions of small tasks that ultimately carry out a larger task.

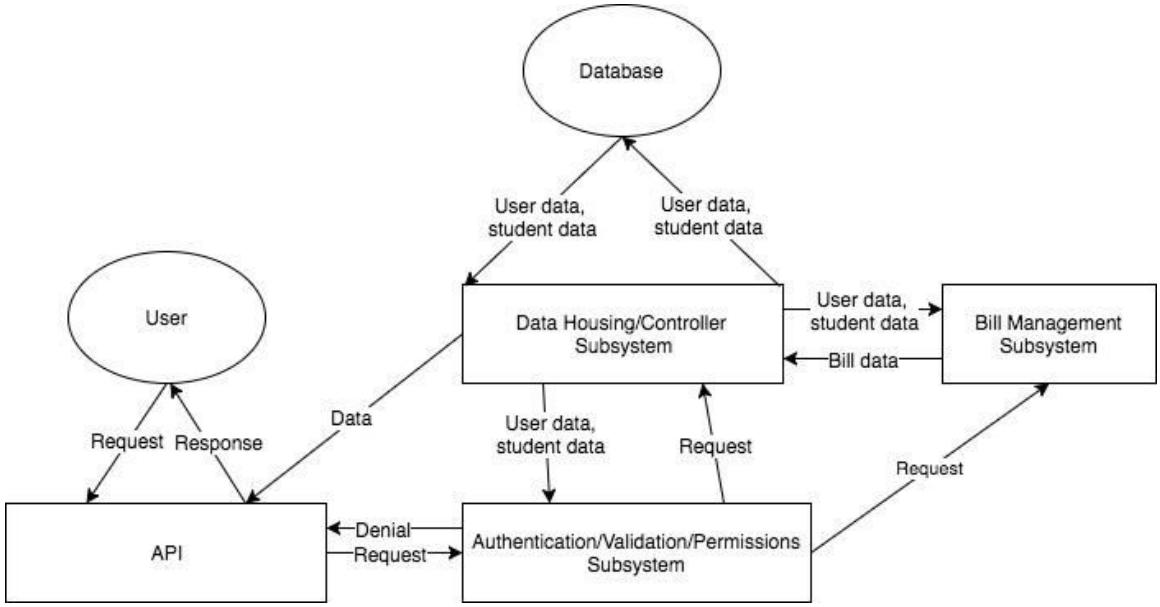
We also considered routing of function calls, to create a parental/hierarchical calling structure, thus it can be seen in our design that requests to the system pass through a series of units before being carried out.

Note: draw.io was used to create or graphs and diagrams.

*2.2 **Environment Overview***

The BILL API will strictly be built in Java 8. We do not have details of the exact environment in which the API will be utilized, but we do know that the user must interact with BILL strictly through the API. Any access to BILL shall be prohibited if the API is not used. Simply put: the user will request through the API, and shall be returned data.

2.3 ***System Architecture***



2.3.1 Top-level system structure of the BILL system

The Billable Interface Linkable Library (BILL) consists of three major subsystems: Authentication/Validation/Permissions Subsystem (AVPS), Data Housing/Controller Subsystem (DHCS), and a Bill Management subsystem (BMS).

Upon an initial API call to BILL, the software will extract and initialize the DHCS with the following data from the database:

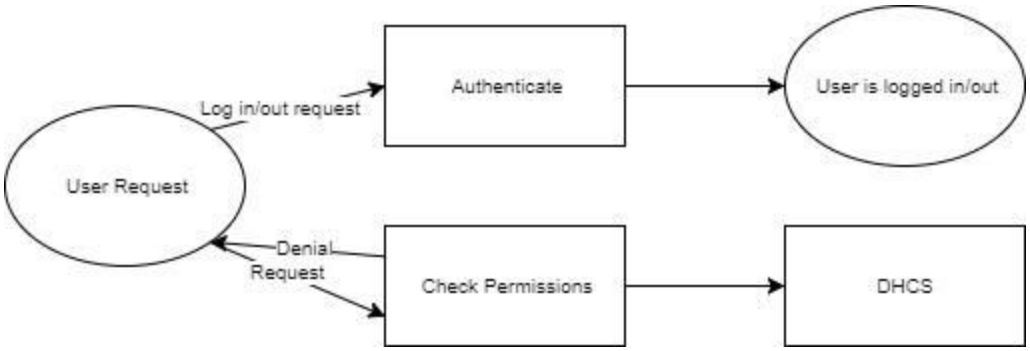
- User data
- Student data

Initializing the DHCS entails internally modeling this data and storing these objects within the Data Housing/Controller Subsystem. Once BILL has initialized data to the DHCS, all subsequent requests and updates to any data in the database will flow be transmitted via the DHCS.

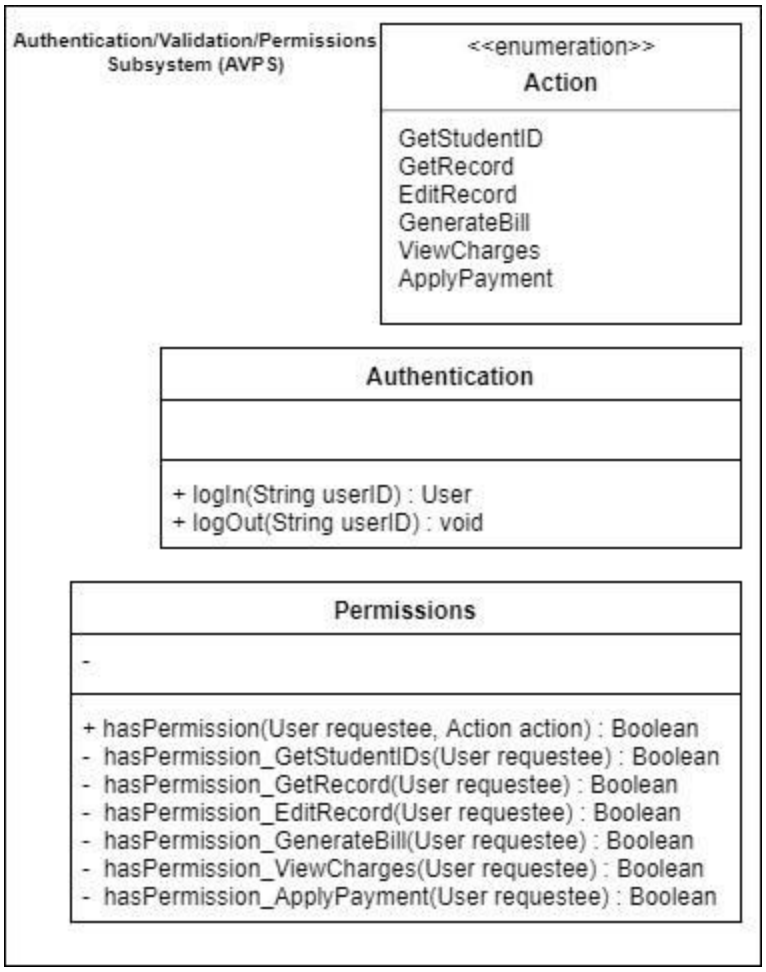
The Authentication/Validation/Permissions Subsystem (AVPS) processes all requests coming from the API to the software and tests the requests for authentication, validation, and appropriate permissions. No dataflow will occur out of the AVPS if a request cannot be validated.

Once a request has been validated by AVPS and if bill calculation or modification is necessary, the Billing Management Subsystem (BMS) gets the user data, student data, and bill data from the DHCS and carries out logic based on this data to calculate/update a bill(s), and returns this [modified] bill to the DHCS. If the bill has been modified, the DHCS then writes the [modified] data to the database.

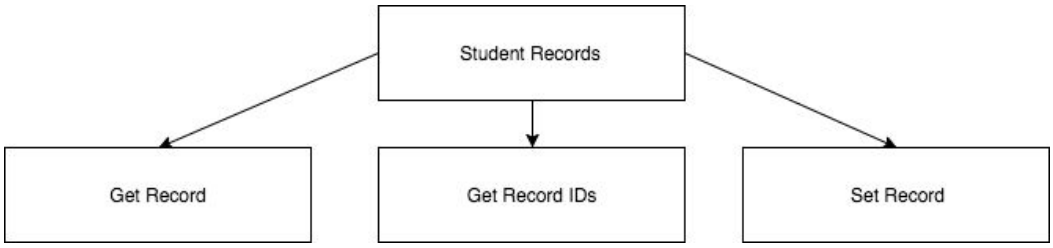
2.3.2 Authentication/Validation/Permissions Subsystem (AVPS)



The AVPS is responsible for processing requests to log the user in/out, and ensures a given user has permissions to make a given call/request (validates permissions) and validates input values. If a call/request passes permission validation, then the request is sent to the Data Housing and Control Subsystem.

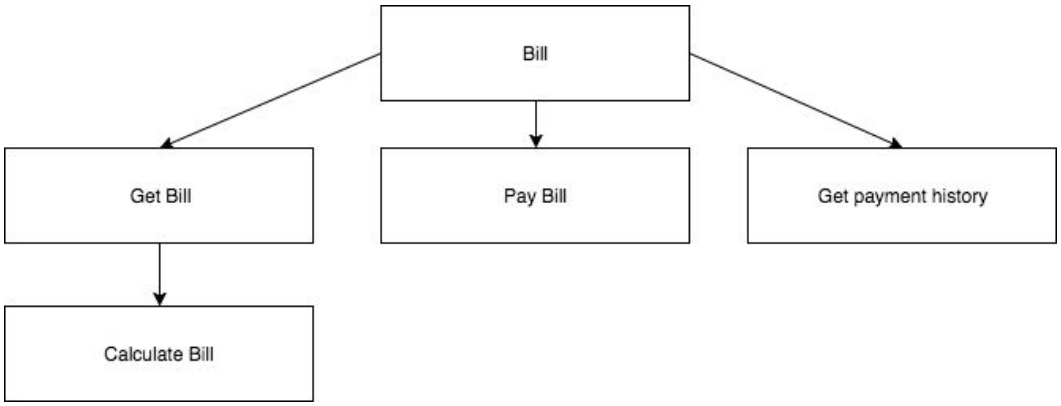


2.3.3 Student Records Subsystem



The student record subsystem consists of getting records, settings records, and retrieving the ID's of all student records. Setting records consist of editing the different fields of a record.

2.3.4 Bill Subsystem

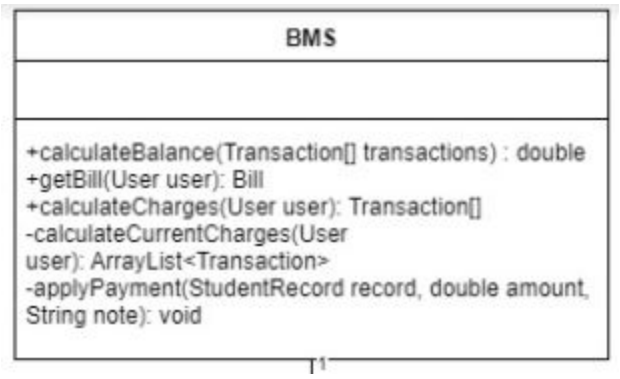


The bill subsystem consists of getting the bill which in turn will calculate the bill. It also consists of paying the bill and viewing payment history.

2.4 Billing Management Subsystem

The Billing Management Subsystem (BMS) is responsible for carrying out all necessary logic to calculate a student bill, generating a Bill, and querying charges/payments. In the case that a bill is requested to be generated, the BMS performs logic based on the attributes of the supplied user to calculate charges/fees. For example, if the provided student is an out of state student, then an out of state charge may be accumulated; the BMS handles determining if this charge (and others) should be applied for the provided student. Additionally, a request may be made to view historical data, in which case a date range is provided and historical charges/payment data is returned.

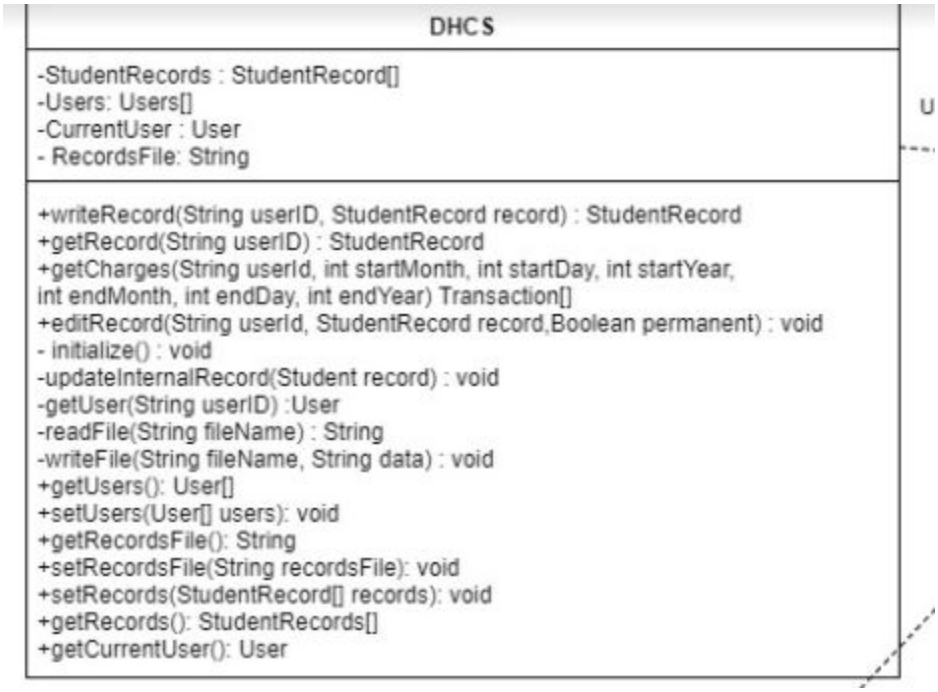
Since requests are only passed to the BMS *after* the request has passed permission validation, it can be assumed that all requests coming into the BMS are valid requests, and may be executed without concern of information leakage.



2.5 Data Housing/Controller Subsystem (DHCS)

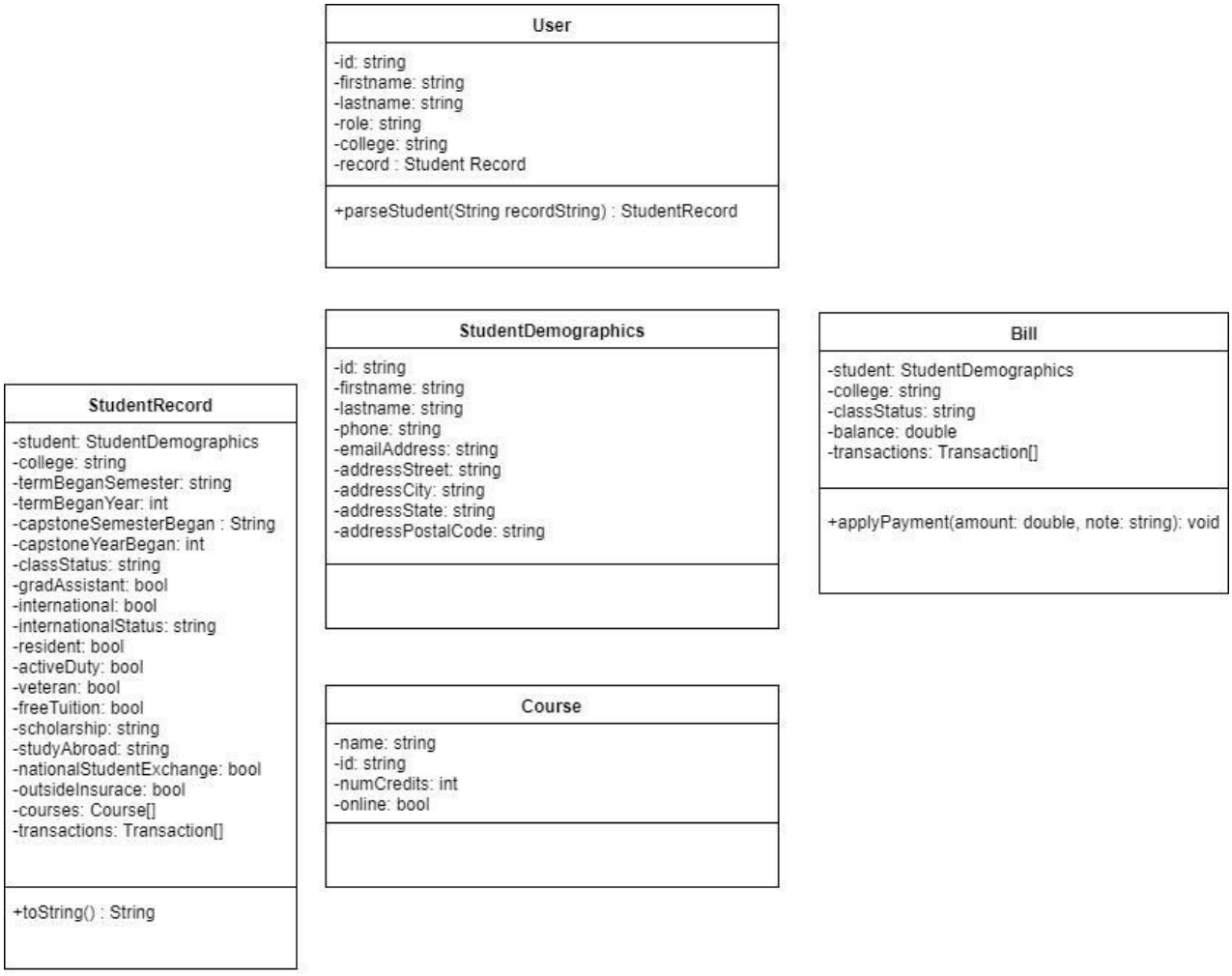
The Data Housing Controller Subsystem (DHCS) is responsible for all input/output to and from the database, as well as maintaining an internal/runtime version of modified student records. Public functionality includes reading writing a student record, and getting charges from the database and editing a record. This functionality entails the need for private (helper) functionality of updating internal records, getting user objects from the database, reading a file, and writing a file.

Since requests are only passed to the DHCS *after* the request has passed permission validation, it can be assumed that all requests coming into the DHCS are valid requests, and may be executed without concern of information leakage or improper modification.



2.6 Abstract Data Types

The following data types are used to maintain user information and billing information. This follows OO techniques.



2.7 Constraints and Assumptions

1. All user authentication is provided by a 3rd party software and we must assume that all requests to access the system have been previously authenticated via username and password. We shall receive the user ID of the authenticated user.
2. Our API will be accessed by client side applications, most of which now consume data in JSON format. We are constrained to input and output JSON formatted data. The API will utilize Google's GSON library for JSON parsing (http://code.google.com/p/google-gson/)

3. Our API will be built using Java 8

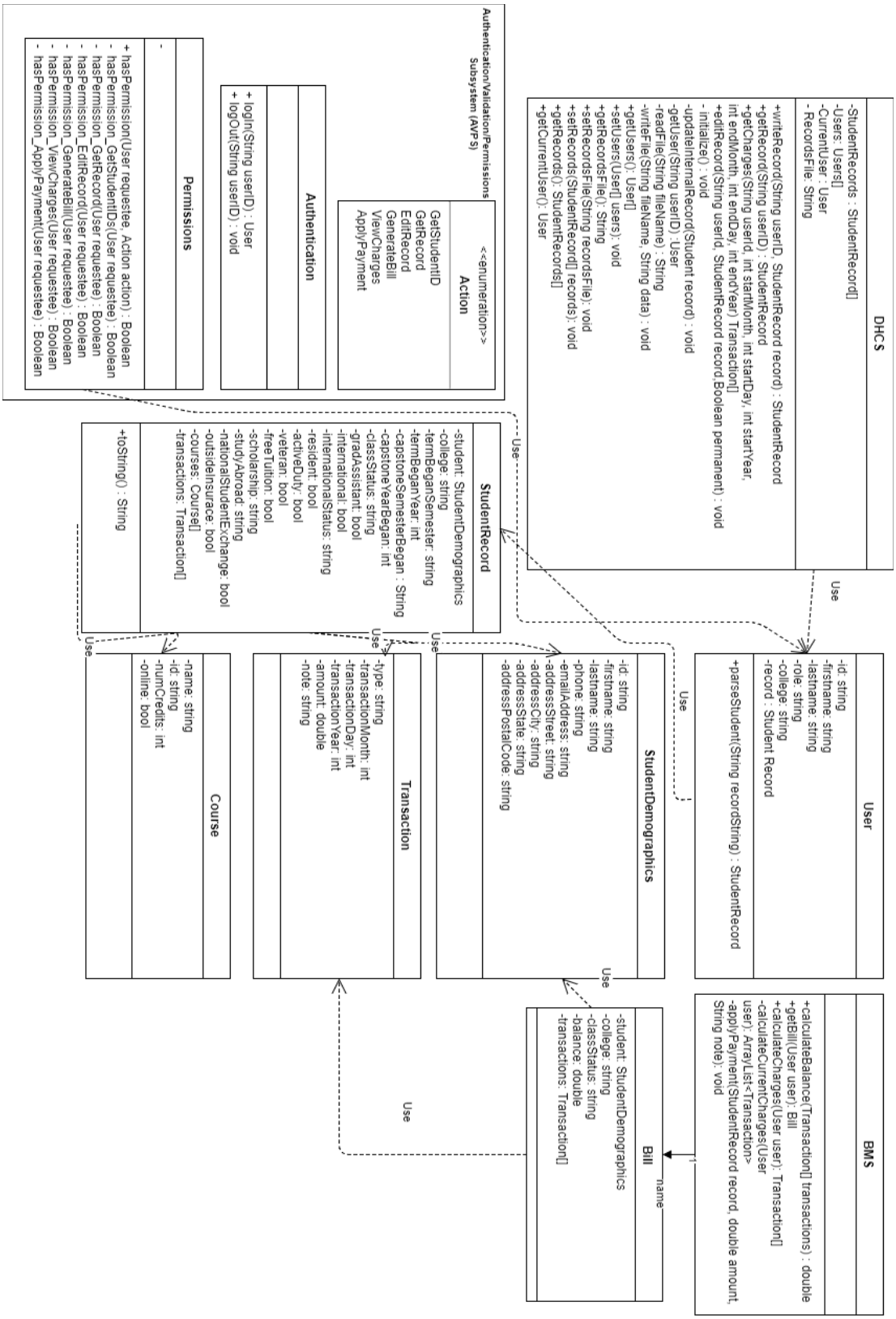
3 Interfaces and Data Stores

This section describes the interfaces into and out of the system as well as the data stores you will be including in your system.

3.1 System Interfaces

There are 2 other external systems that BILL Interfaces with. Bill will interface with a 3rd party authentication system that authenticates users upon logging in. It will also interface with two databases: a user database, and a student record database.

4 Structural Design


DHCS
-StudentRecords : StudentRecord[]
-Users: Users[]
-CurrentUser : User
- RecordsFile: String
+writeRecord(String userID, StudentRecord record) : StudentRecord
+getRecord(String userID) : StudentRecord
+getCharges(String userId, int startMonth, int startDay, int startYear, int endMonth, int endDay, int endYear) Transaction[]
+editRecord(String userId, StudentRecord record,Boolean permanent) : void
- initialize() : void
-updateInternalRecord(Student record) : void
-getUser(String userID) :User
-readFile(String fileName) : String
-writeFile(String fileName, String data) : void
+getUsers(): User[]
+setUsers(User[] users): void
+getRecordsFile(): String
+setRecordsFile(String recordsFile): void
+setRecords(StudentRecord[] records): void
+getRecords(): StudentRecords[]
+getCurrentUser(): User
Use
User
-id: string
-firstname: string
-lastname: string
-role: string
-college: string
-record : Student Record
+parseStudent(String recordString) : StudentRecord
BMS
+calculateBalance(Transaction[] transactions) : double
+getBill(User user): Bill
+calculateCharges(User user): Transaction[]
-calculateCurrentCharges(User user): ArrayList<Transaction>
-applyPayment(StudentRecord record, double amount, String note): void
1
name
Bill
-student: StudentDemographics
-college: string
-classStatus: string
-balance: double
-transactions: Transaction[]
Use
StudentDemographics
-id: string
-firstname: string
-lastname: string
-phone: string
-emailAddress: string
-addressStreet: string
-addressCity: string
-addressState: string
-addressPostalCode: string
Use
StudentRecord
-student: StudentDemographics
-college: string
-termBeganSemester: string
-termBeganYear: int
-capstoneSemesterBegan : String
-capstoneYearBegan: int
-classStatus: string
-gradAssistant: bool
-international: bool
-internationalStatus: string
-resident: bool
-activeDuty: bool
-veteran: bool
-freeTuition: bool
-scholarship: string
-studyAbroad: string
-nationalStudentExchange: bool
-outsideInsurance: bool
-courses: Course[]
-transactions: Transaction[]
+toString() : String
Use
Transaction
-type: string
-transactionMonth: int
-transactionDay: int
-transactionYear: int
-amount: double
-note: string
Use
Course
-name: string
-id: string
-numCredits: int
-online: bool
Use
Use
Use
Use
Use
Authentication/Validation/Permissions Subsystem (AVPS)
<<enumeration>>
Action
GetStudentID
GetRecord
EditRecord
GenerateBill
ViewCharges
ApplyPayment
Authentication
+ logIn(String userID) : User
+ logOut(String userID) : void
Permissions
-
+ hasPermission(User requestee, Action action) : Boolean
- hasPermission_GetStudentIDs(User requestee) : Boolean
- hasPermission_GetRecord(User requestee) : Boolean
- hasPermission_EditRecord(User requestee) : Boolean
- hasPermission_GenerateBill(User requestee) : Boolean
- hasPermission_ViewCharges(User requestee) : Boolean
- hasPermission_ApplyPayment(User requestee) : Boolean

4.1.1 Diagram: User

User
-id: string -firstname: string -lastname: string -role: string -college: string -record : StudentRecord
+getUser(): String +parseStudent(String recordString)

4.1.2 Class: User

- Purpose: *To model the aspects of a user that needs to access student record data and student billing data*
- Constraints: *None*
- Persistent: *Yes*

4.1.3 Attributes

1. Attribute: *id*
   Type: *string*
   Description: *Stores the unique id of the User*
   Constraints: *Unique*

2. Attribute: *firstname*
   Type: *string*
   Description: *Stores the first name of the User*
   Constraints: *None*

3. Attribute: *lastname*
   Type: *string*
   Description: *Stores the last name of the User*
   Constraints: *None*

4. Attribute: *role*
   Type: *string*
   Description: *Stores the role of the User, the main determinant of permissions*
   Constraints: *[student, user]*

5. Attribute: *college*
   Type: *string*
   Description: *Stores the college of the User*
   Constraints: *[CSE, ARTS_SCIENCES, GRADUTE_SCHOOL]*

6. Attribute: *record*
   Type: *StudentRecord*
   Description: *This contains all of the information needed for a student*
   Constraints: *none*

4.1.4 Methods

1. Name: *getUser()*
   Parameters: *None*
   Return: *User ID*
   Description: *Returns the ID of the current user*

2. *Name: parseStudent(String recordString)*
   Parameters:
   - recordString: String, contains the information to be parsed into a record

   Return: *StudentRecord record*
   Description: *returns a student record created by parsing the input String*

4.2.1 Diagram: Bill

Bill
-student: StudentDemographics -college: string -classStatus: string -balance: double -transactions: Transaction[]
+getPaymentHistory(startMonth: int, startDay: int, startYear: int, endMonth: int, endDay: int, endYear: int): Transactions[] +applyPayment(amount: double, note: string): void

4.2.2 Class: Bill

- Purpose: *To model the aspects of a bill that is associated to a student*
- Constraints: *None*
- Persistent: *Yes*

4.2.3 Attributes

1. Attribute: *balance*
   Type: *double*
   Description: *Balance of the bill, summation of all charges*
   Constraints: *Positive*

2. Attribute: *transactions*
   Type: *Transaction[]*
   Description: *List of all transactions associated with the bill*
   Constraints: *None*

4.2.4 Methods

1. Name: *getPaymentHistory()*
   Parameters:
   - *startMonth: int, month of start date to filter transactions*
   - *startDay: int, day of start date to filter transactions*
   - *startYear: int, year of start date to filter transactions*
   - *endMonth: int, month of end date to filter transactions*
   - *endDay: int, day of start end to filter transactions*
   - *endYear: int, year of start end to filter transactions*

   Return: *List of transactions that were created in between or on the startDate and endDate*
   Description: *Will filter all transactions attached to the bill by the startDate and endDate*

2. Name: *applyPayment()*
   Parameters:
   - *amount: double, amount to pay on bill total*
   - *note: string, any comments to be made for payment*

   Return: *None*
   Description: *Provided startDate and endDate must be in order and if endDate is greater than current date then the method will return all transactions from the startDate and current date. If the amount is greater than the bill total or is negative then an error will be thrown.*

4.3.1 Diagram: StudentDemographics

StudentDemographics
-id: string -firstname: string -lastname: string -phone: string -emailAddress: string -addressStreet: string -addressCity: string -addressState: string -addressPostalCode: string

4.3.2 Class: StudentDemographics

- Purpose: *To model the aspects of a student's demographic and contact information*
- Constraints: *None*
- Persistent: *Yes*

4.3.3 Attributes

1. Attribute: *id*
   Type: *string*
   Description: *ID of student*
   Constraints: *Unique*

2. Attribute: *firstname*
   Type: *string*
   Description: *First name of student*
   Constraints: *None*

3. Attribute*: lastname*
   Type: *string*
   Description: *Last name of student*
   Constraints: *None*

4. Attribute*: phone*
   Type: string
   Description: *Phone number of student*
   Constraints: *Must be in format XXX-XXX-XXXX*

5. Attribute*: emailAddress*
   Type: *string*
   Description: *Email address of student*
   Constraints: *Must be valid email*

6. Attribute*: addressStreet*
   Type: *string*
   Description: *Address of student*
   Constraints: *None*

7. Attribute*: addressCity*
   Type: *string*
   Description: *City of student*
   Constraints: *None*

8. Attribute*: addressState*
   Type: *string*
   Description: *State of student*
   Constraints: *None*

9. Attribute*: addressPostalCode*

Type: *string*
Description: *Postal code of student*
Constraints: *Must be in format XXXXX*

4.4.1 Diagram: StudentRecord

StudentRecord
-student: StudentDemographics -college: string -termBegan: Term -classStatus: string -gradAssistant: bool -international: bool -internationalStatus: string -resident: bool -activeDuty: bool -veteran: bool -freeTuition: bool -scholarship: string -studyAbroad: string -nationalStudentExchange: bool -outsideInsurace: bool -courses: Course[] -transactions: Transaction[]
+getTransactionPeriod(int startMonth, int startDay, int startYear, int endMonth, int endDay, int endYear) +toString() : String

4.4.2 Class: StudentRecord

- Purpose: *To model the aspects of a student*
- Constraints: *None*
- Persistent: *Yes*

4.4.3 Attributes

1. Attribute: *student*
   Type: *StudentDemographics*
   Description: *Student demographic and contact information*
   Constraints: *None*

2. Attribute: *college*
   Type: *string*
   Description: *Student college*
   Constraints: *[CSE, ARTS_SCIENCES, GRADUTE_SCHOOL]*

3. Attribute*: termBegan*
   Type: *Term*
   Description: *Term the student began*
   Constraints: *On or before current term*

4. Attribute*: classStatus*
   Type: *string*
   Description: *Class status of student*
   Constraints: *None*

5. Attribute*: gradAssistant*
   Type: *bool*
   Description: *Is the student a graduate assistant*
   Constraints: *None*

6. Attribute*: international*
   Type: *bool*
   Description: *Is the student an international student*
   Constraints: *None*

7. Attribute*: internationalStatus*
   Type: *string*
   Description: *Student international status*
   Constraints: *“NONE” if international is false*

8. Attribute*: resident*
   Type: *bool*
   Description: *Is student an in-state resident*
   Constraints: *None*

9. *Attribute: activeDuty*
   Type: *bool*
   Description: *Is student active military*
   Constraints: *Cannot be true if veteran is true*

10. Attribute*: veteran*
    Type: *bool*
    Description: *Is student a military veteran*
    Constraints: *Cannot be true if activeDuty is true*

11. Attribute*: freeTuition*
    Type: *bool*
    Description: *Is student receiving free tuition*
    Constraints: *None*

12. Attribute*: scholarship*
    Type: *string*
    Description: *Student scholarship*
    Constraints: *[WOODROW, DEPARTMENTAL, GENERAL, ATHLETIC, SIMS, NONE]*

13. Attribute*: studyAbroad*
    Type: *bool*
    Description: *Is student studying abroad*
    Constraints: *Cannot be true if nationalStudentExchange is true*

14. Attribute*: nationalStudentExchange*
    Type: *bool*
    Description: *Is student participating in a national student exchange*
    Constraints: *Cannot be true if studyAbroad is true*

15. Attribute*: outsideInsurance*
    Type: *bool*
    Description: *Is student using outside insurance*
    Constraints: *None*

16. Attribute*: courses*
    Type: *Course[]*
    Description: *Student's current courses*
    Constraints: *None*

17. Attribute*: transactions*
    Type: *Transaction[]*
    Description: *Student's bill payment transactions*
    Constraints: *None*

4.4.4 Methods

1. *Name: toString()*
   Parameters: none
   return: a String containing the information in the student record
   Description: This will convert all of the information in a student record into a string

4.5.1 Diagram: Term

Term
-semester: string -year: int

4.5.2 Class: Term

- Purpose: *To model the aspects of a school term*
- Constraints: *None*
- Persistent: *Yes*

4.5.3 Attributes

1. Attribute: *semester*
   Type: *string*
   Description: *Name of semester*
   Constraints: *["FALL", "SPRING", "SUMMER"], Current semester or before if year is same as current*
2. Attribute: year
   Type: *string*
   Description: *Year of semester*
   Constraints: *Must be current year or before*

4.6.1 Diagram: Course

Course
-name: string -id: string -numCredits: int -online: bool

4.6.2 Class: Course

- Purpose: *To model the aspects of a course that student is taking*
- Constraints: *None*
- Persistent: *Yes*

4.6.3 Attributes

1. Attribute: *name*
   Type: *string*
   Description: *Course name*
   Constraints: *None*

2. Attribute: *id*
   Type: *string*
   Description: *Course ID*
   Constraints: *Must be in format XXX XXX*

3. Attribute: *numCredits*
   Type: *int*
   Description: *Number of credits the course awards*
   Constraints: *0-4*

4. Attribute: *online*

Type: *bool*
Description: *Is the course online*
Constraints: *None*

4.7.1 Diagram: Transaction

Transaction
-type: string -transactionMonth: int -transactionDay: int -transactionYear: int -amount: double -note: string

4.7.2 Class: Transaction

- Purpose: *To model the aspects of a transaction on a student's bill*
- Constraints: *None*
- Persistent: *Yes*

4.7.3 Attributes

1. Attribute: type
   Type: *string*
   Description: *Transaction type*
   Constraints: *["CHARGE", "PAYMENT", "REFUND"]*

2. Attribute: *transactionMonth*
   Type: *int*
   Description: *Transaction month*
   Constraints: *1-12*

3. Attribute: transactionDay
   Type: *int*
   Description: *Transaction day*
   Constraints: *1-31 and other special circumstances*

4. Attribute: *transactionYear*
   Type: *int*
   Description: *Transaction year*
   Constraints: *Must be in format XXXX, current year of before*

5. Attribute: *amount*
   Type: *double*
   Description: *Transaction amount*

Constraints: *Positive*

6. Attribute: *notes*
   Type: *string*
   Description: *Transaction notes*
   Constraints: *None*

5 Detailed Class and Method Definitions

5.1 Class AVPS

- Overview: this class handles Logging In/Out (Authentication), ensures input is well-formed (Validation), and ensures a given user has permissions to make a given call/request (Permissions)
- Parent Class: none
- Attributes: none
- Methods:
    - public boolean hasPermission(User requestee, Action action)
        - Description: Determines if a user has permission to carry out an action against another (possibly the same) user
        - Parameters: User for which the action is being carried against, action being performed
        - Return value: true if user has permission for action against requestee
    - private boolean hasPermission_LogIn(User requestee)
        - Description: The user which is requesting permissions for logging in
        - Parameters: User for which the action is being carried against
        - Return value: true if user has permission for action against requestee
    - private boolean hasPermission_LogOut(User requestee)
        - Description: The user which is requesting permissions for logging out
        - Parameters: User for which the action is being carried against
        - Return value: true if user has permission for action against requestee
    - private boolean hasPermission_ApplyPayment(User requestee)
        - Description: Determines if a user has permission to apply payment for another user
        - Parameters: User for which the action is being carried against
        - Return value: true if user has permission for action against requestee
    - private boolean hasPermission_ViewCharges(User requestee)
        - Description: Determines if a user has permission to view charges for another user
        - Parameters: User for which the action is being carried against
        - Return value: true if user has permission for action against requestee
    - private boolean hasPermission_GenerateBill(User requestee)
        - Description: Determines if a user has permission to get a bill for another user
        - Parameters: User for which the action is being carried against
        - Return value: true if user has permission for action against requestee
    - private boolean hasPermission_EditRecord(User requestee)
        - Description: Determines if a user has permission to edit a student record for another user
        - Parameters: User for which the action is being carried against

        - Return value: true if user has permission for action against requestee
    - private boolean hasPermission_GetRecord(User requestee)
        - Description: Determines if a user has permission to get student record for another user
        - Parameters: User for which the action is being carried against
        - Return value: true if user has permission for action against requestee
    - private boolean hasPermission_GetStudentId(User requestee)
        - Description: Determines if a user has permission to get student id for another user
        - Parameters: User for which the action is being carried against
        - Return value: true if user has permission for action against requestee
    - private boolean validateRecord(StudentRecord record)
        - Description: Determine if a student record is valid
        - Parameters: Record that is to be edited
        - Return value: String that consists of error, empty string signifies no error
    - private boolean isValid<field-name>(<type> <field-name>)
        - Description: Determine if a field is valid before allowing edit of that field in record
        - Parameters: Field to be edited
        - Return value: true if valid edit
        - Repeat:
            - Transactions
            - Transaction
            - Courses
            - StudyAbroad
            - Scholarship
            - InternationalStatus
            - Status
            - Term
            - State
            - College
            - Email
    - private boolean isNotNull(String aString)
        - Description: Determine if string is not null
        - Parameters: String to validate
        - Return value: true if not null

5.2 Class DHCS

- Overview: Handles all File I/O and errors, and encapsulates the data into objects that is represented in database files
- Parent Class: none
- Attributes:
    - public StudentRecord[] studentRecords
        - Internal representation of the Student Records that are in the database
    - public User[] users
        - Internal representation of the Users that are in the database
    - public User currentUser
        - Holds current *authenticated* user or null if there is no authenticated user
    - public String recordsFile
        - File that records were read from so is later know what file to write edits to

- Methods:
  - public User getUsers(User[] users)
    - Description: gets users from database
    - Parameters: none
    - Returns: List of users
    - Logic: Get users that were set with setUsers
  - public User setUsers(User[] users)
    - Description: set users to database
    - Parameters: users
    - Returns: none
    - Logic: Set users in local database
  - public User getRecordsFile()
    - Description: Get name/path of records file
    - Parameters: none
    - Returns: records file
  - public User setRecordsFile(String recordsFile)
    - Description: Set name/path of records file
    - Parameters: records file
    - Returns: none
  - public StudentRecord getRecords()
    - Description: returns all student records
    - Parameters: none
    - Returns: student records
    - Logic: Get records that were set with setRecords
  - public StudentRecord setRecords(StudentRecords records)
    - Description: Sets all student records to database
    - Parameters: records
    - Returns: none
    - Logic: Set student records in local database
  - public User getCurrentUser()
    - Description: Get user that is currently logged in
    - Parameters: none
    - Returns: user
  - public User getUser(String userID)
    - Description: gets a user from database
    - Parameters: the user ID
    - Returns: user object that corresponds to user ID
    - Logic: Gets users from users attribute and searches for user with user id userID
  - public void writeRecord(String userID, StudentRecord record) throws Exception
    - Description: writes a given record to the database
    - Parameters: student ID, student record
    - Returns: none
    - Logic: Essentially calls writeFile(fileName, record.toString())
  - public StudentRecord getRecord(String userID) throws Exception
    - Description: returns a student record for a given userID
    - Parameters: the user ID
    - Returns: student record corresponding to user ID
    - Logic: Get records from studentRecords attribute and processes the returned data

- public Transaction[] getCharges(String userId, int startMonth, int startDay, int startYear, int endMonth, int endDay, int endYear)
  - Description: gets a list of student charges for a given date range
  - Returns: list of Transactions for the given date range
  - Logic:
    - Validate date range
    - StudentRecord record = getRecord(userId)
    - // filters down *record.transactions[]*, converts the filtered down *record.transactions[]* to a *Bill*, and returns that *Bill*
- private void updateInternalRecord(Student record)
  - Description: Updates DHCS.StudentRecords with record
  - Parameters: the record
  - Returns: none

5.3 Class Billing

- Overview: handles bill calculation logic
- Parent Class: none
- Attributes: none
- Methods:

  - public static double calculateBalance(Transaction[] transactions)
    - Description: Calculates balance by summing all payments and that from the sum of all charges for a given transaction array.
    - Parameters:
      - transactions Transactions for which the balance is being calculated
    - Returns: the balance
    - Logic: [see description]

  - public static Bill getBill(User user)
    - Description: Generates a Bill, comprised of both applicable current charges and historical charges for a given user by calling calculateCharges(...).
    - Parameters:
      - user User for which the bill is being calculated
    - Returns: Bill object
    - Logic: Simply calls the parameterized constructor for Bill and calculateCharges(...).
  - private ArrayList<Transaction> calculateCurrentCharges(User user)
    - Description: Gets all applicable fees based on logic coming from the USC fee schedule, and combines them with the fees already on the student record
    - Logic: call mergeBillingInfo(applicableFees, DCHS.getCharges(...), stud)
  - public Transaction[] calculateCharges(User user)
    - Description: Gets all applicable fees based on logic coming from the USC fee schedule (by calling calculateCurrentCharges), and combines them with the fees already on the student record
    - Parameters: User for which bill is being generated
    - Returns: Transaction[] representing merge of current charges and historical charges

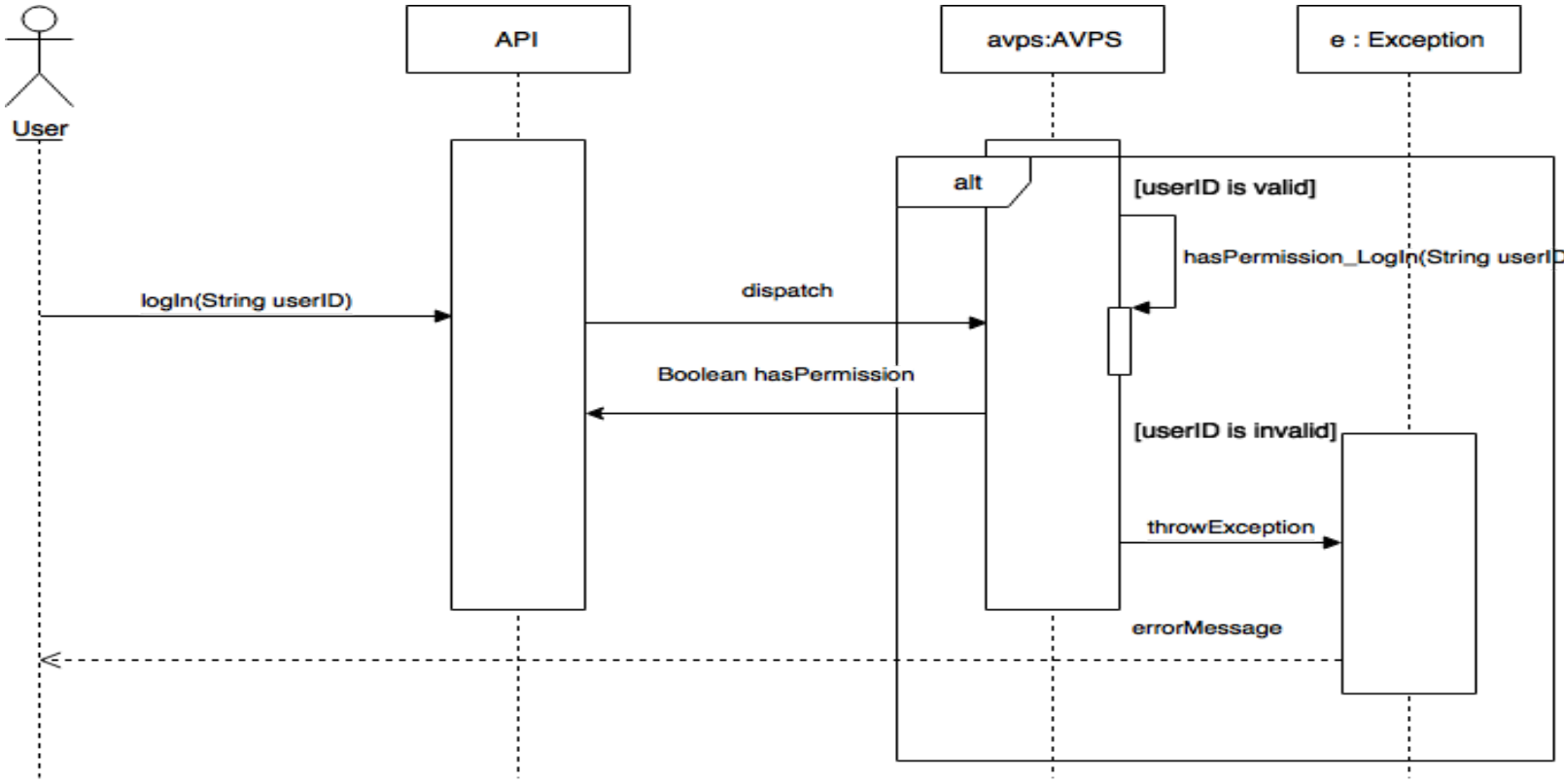
- public static void applyPayment(StudentRecord record, double amount, String note) throws InvalidPaymentException
  - Description: Applies payment to a given student record.
  - Parameters:
    - record The student record for which the payment should be applied
    - amount Amount of payment
    - note Description of payment
  - Returns: nothing

6 Dynamic Model

6.1 Scenarios

6.2

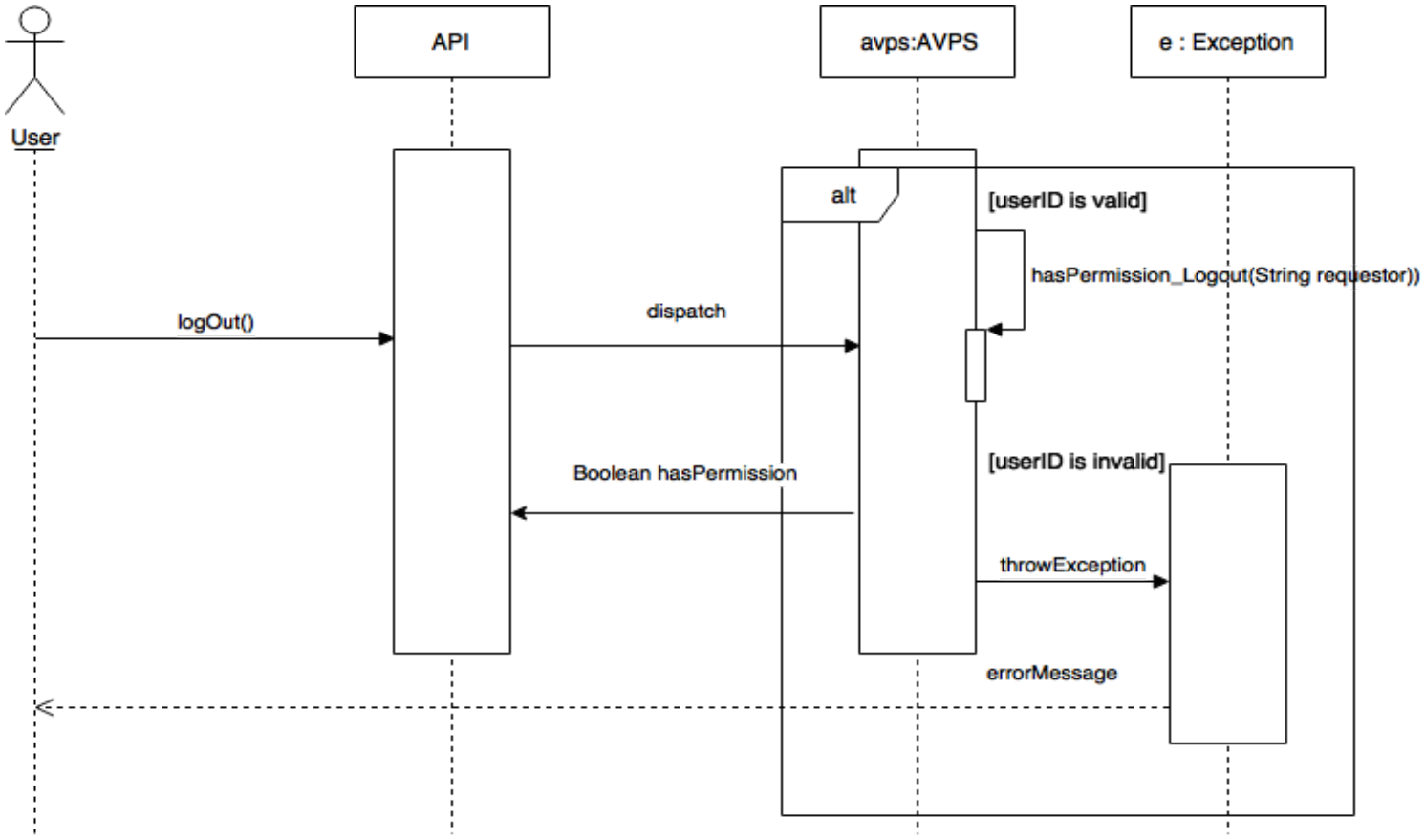
1. Log In
   a. This scenario describes the use case where a user would make a call to authenticate and login with the BILL api. The user would first attempt to login using BILL, and their credentials would be validated by a third party provider.



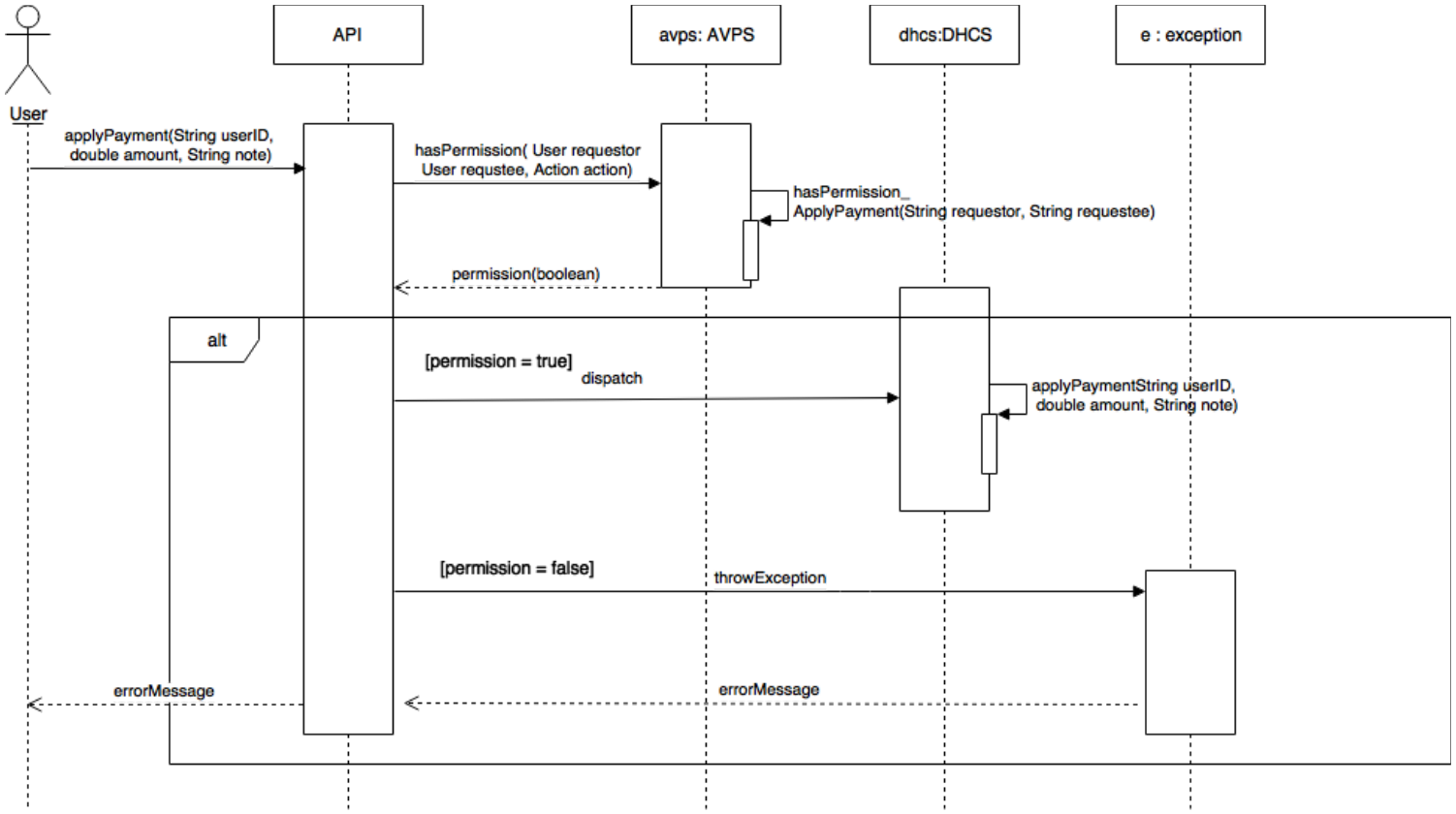
   b.
2. Log Out

a. This scenario describes the use case where a user would make a call to logout with BILL.

b.

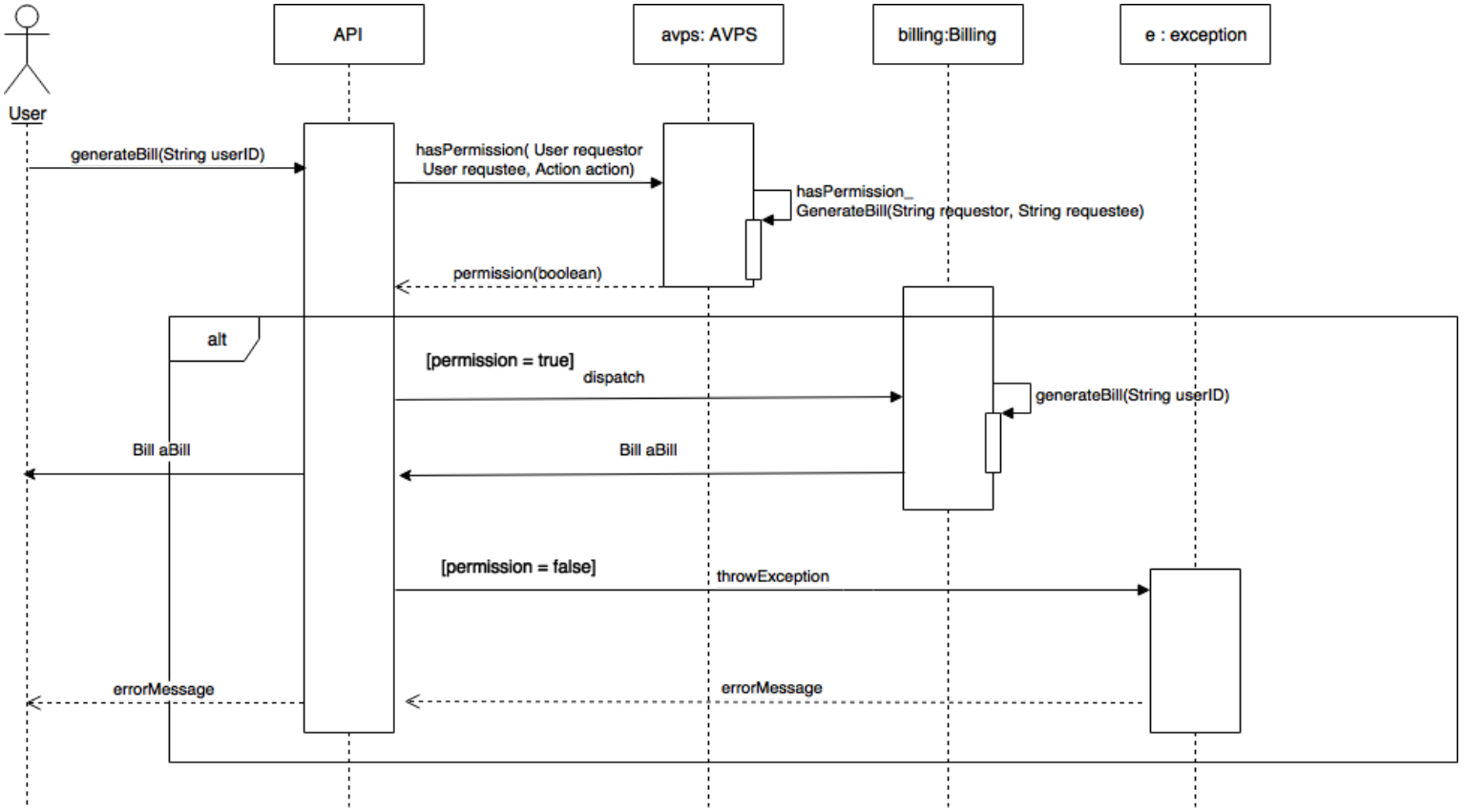


3. Pay Bill
   a. This scenario describes the use case where a user would make a call to pay a student bill with BILL.

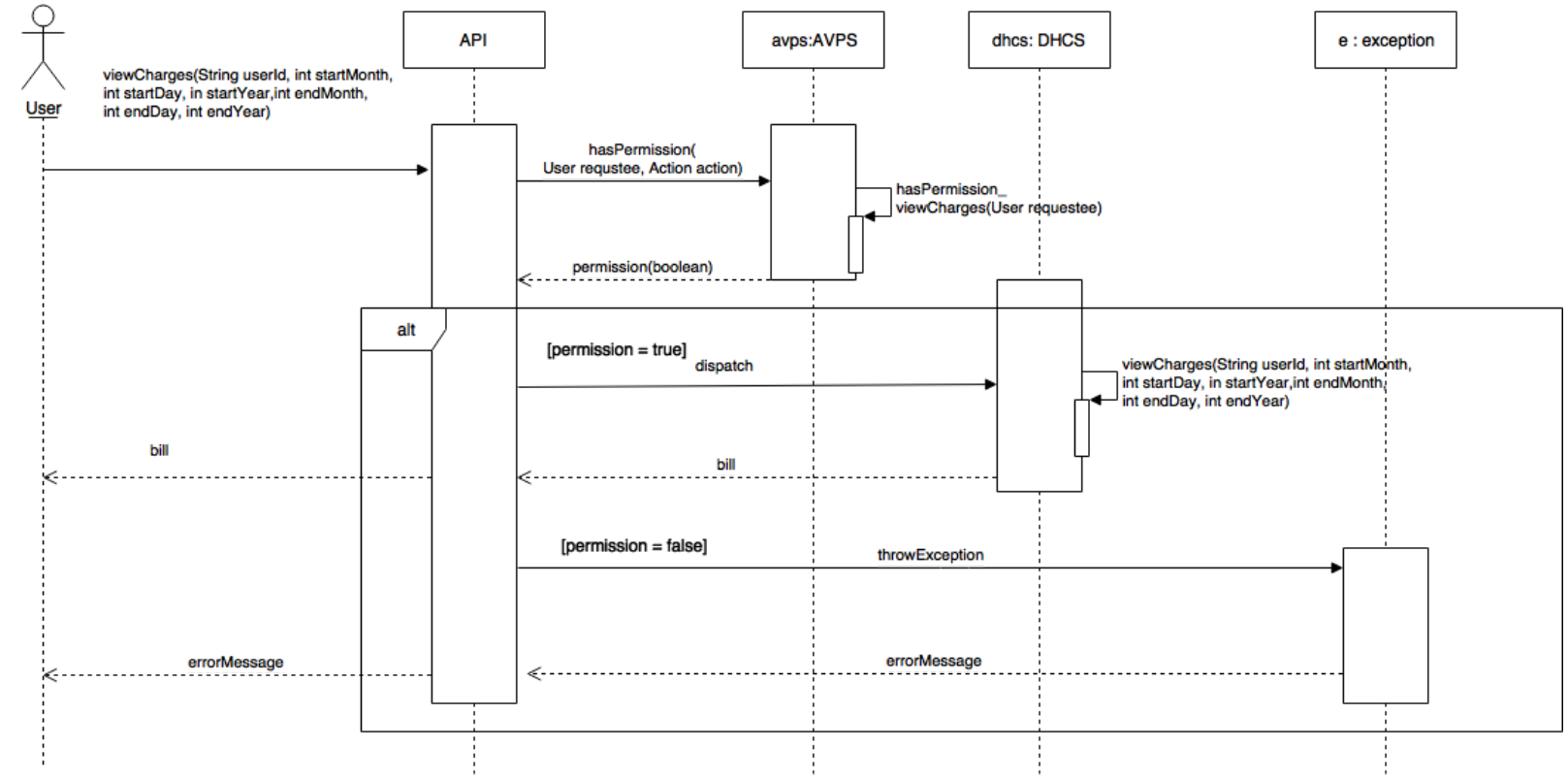


b.

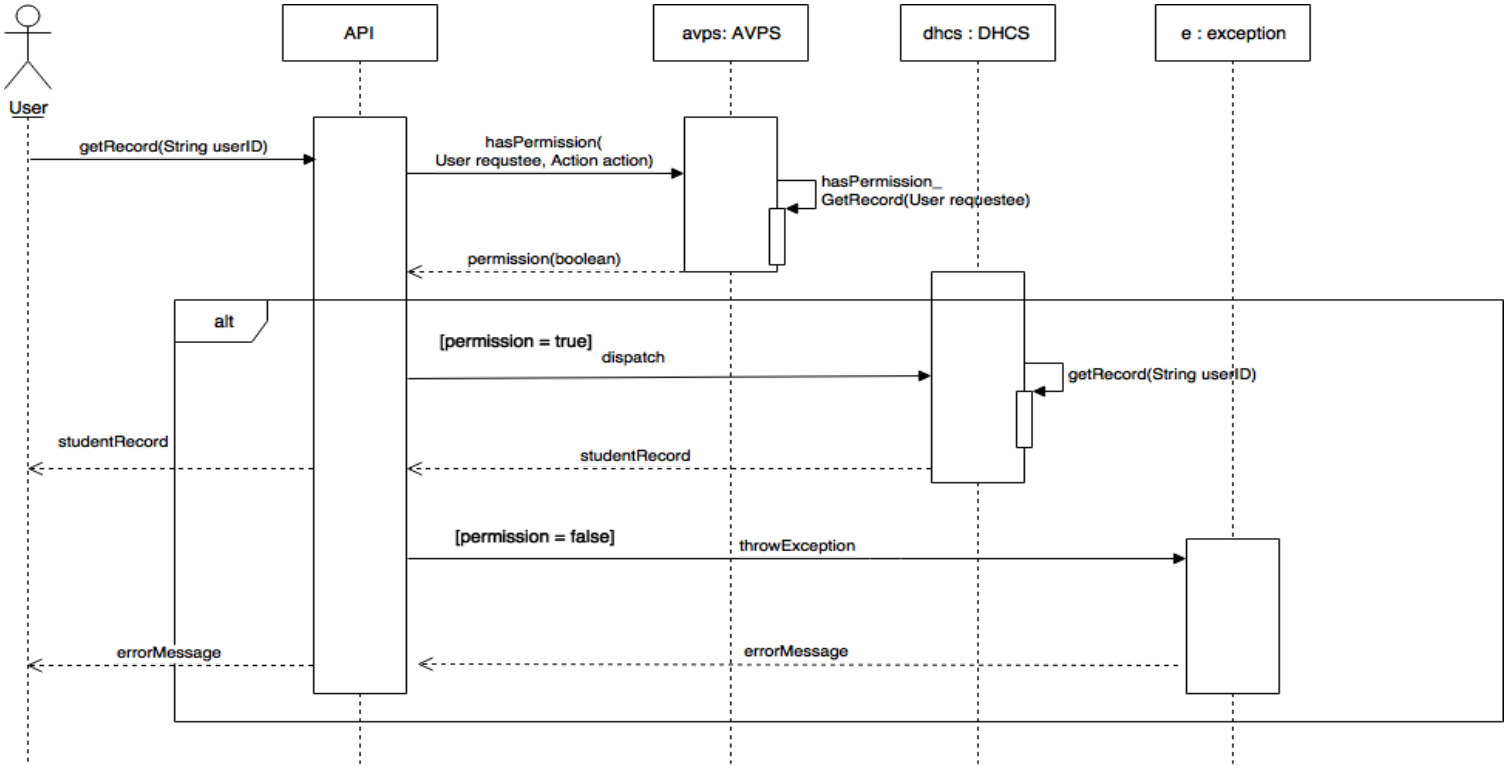
4. Generate Bill
    a. This scenario describes the use case where a user would make a call to view a student bill with BILL.



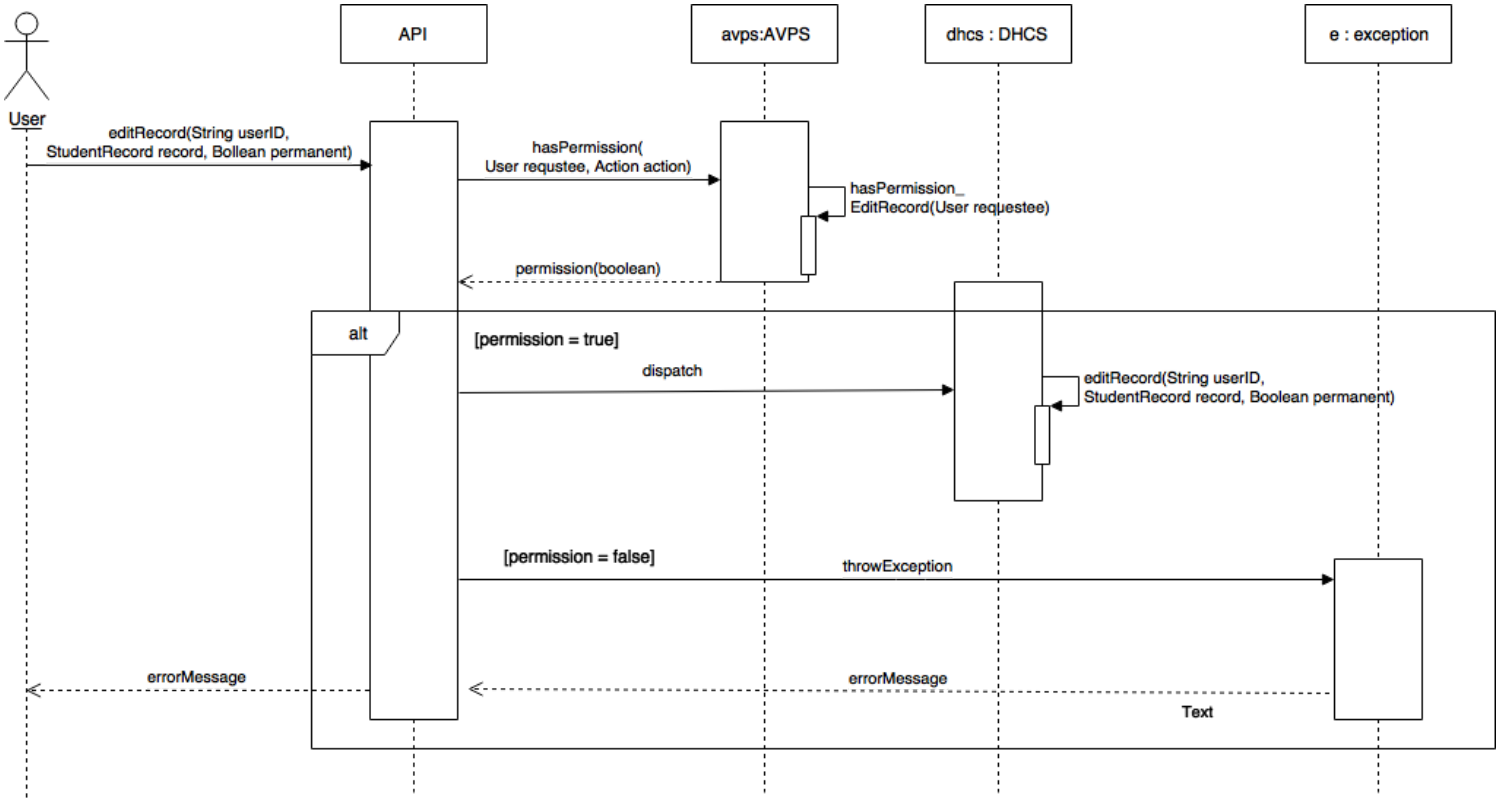
5. View Payment History
    a. This scenario describes the use case where a student user, or an admin user with correct permissions, would make a call to view a student payment history.
    b.



6. View Student Record
    a. This scenario describes the use case where a logged in user would make a call to view a student record.
    b.



7. Edit Student Record
    a. This scenario describes the use case where a logged in admin user with correct permissions would make a call to edit a student record.
    b.



7 Non-functional requirements

The BILL API system shall operate in a manner that does not cause the user to wait for an extended amount of time. We do not have exact quantifiers on completion times just yet, since we have yet to receive these requirements from the customer. However, it is likely that all BILL API requests should likely complete in much less than 1 second.

Over time, it is possible that the database structure will change. Development will be required if this occurs. Additionally, administrators may eventually require more ability to modify student records. The design will need to validate these calls, and carry out these calls which is a simple addition in the AVPS and DHCS subsystems.

8 Supplementary Documentation

In section 4 we are purposely ignoring the inclusion of getters and setters in class methods. It can be assumed that every attribute has its own getters and setter method.