USE CASES (FULLY DRESSED)
==

Purchasing a Ticket

Scope: CharlestonEvents application

Level: User goal

Primary Actor: User

Stakeholders and Interests:

- **User:** Wants to be able to purchase the ticket with minimal effort. Wants proof of purchase to confirm ticket purchase.
- **Government Tax Agencies:** Wants to collect tax from every sale. May be multiple agencies, such as national, state, and county.
- **Company:** Wants to be able to update the events available such as adding or deleting an event**.**
- **Payment Authorization Service:** Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.

Preconditions: User is logged in and must have a form of payment, User already selected an event from events.

Postconditions: Users are able to purchase a ticket. Taxes are correctly calculated. Receipt is generated.

Main Success Scenario:

1. User logs in to the application
2. User finds the event
3. System presents total price with taxes calculated
4. User pays and System handles payment
5. System updates the amount of tickets available
6. System emails the receipt to the User
7. System adds the event to User's Calendar

Extensions:

***a.** The user is not logged in:

1. User finds the event
2. System presents price without taxes being calculated
3. User tries to purchase the ticket
4. System redirects the user to the login page

***b.** The event has no available tickets:

1. User searches for the event
2. User finds the event
3. System informs the user that there are no more tickets available

***c.** The user leaves the application before confirming purchase:

1. System saves the last page visited by the user
2. When the user returns to the application, system displays the last page the user had visited

***d.** User method of payment has failed:

1. User inserts their method of payment

2. The system submits the payment
3. The form of payment has not been accepted
4. System informs the user
5. System tells the user to try another method of payment
    a. If another payment method has been inserted:
        i. Repeat step 2
            1. If payment method has been accepted:
                a. System emails the user the receipt of payment
            2. If payment method has not been accepted:
                a. Repeat step 3 to 5

Special Requirements:

- User must be at least 18 years of age to purchase a ticket
- Credit authorization response within 30 seconds 90% of the time.
- Must be completed within the Fall Semester of 2022

Technology and Data Variations list:

- The application must run on IOS Platform and Android Platform

Browsing Shows

Name: Browsing Shows
Scope: Browse events
Level: User goal
Primary Actor: USER
Stakeholders / Interest:

- **User:**
The user wants to browse and see all events in Charleston, and be able to select an event and proceed on ticket purchasing window.

- **Event organizer:**

The event organizer could list all events for the following months and may do such a thing by requesting developers to update the app, events can be added or removed by the developer team only upon request form the event organizer .

Preconditions:

User is logged in and must have a form of payment to be able to view the events.

Postconditions: User has browsed all events and selected the one particular event, and moved on to the purchasing window.

Main Success Scenario:

8. User searches for an event .
9. User finds the event .
10. User can chose the event
11. User can see a brief description of the event such as date and title.
12. Selected events will be added to the purchasing basket(window).

Extensions:

***a.** The user is not logged in:

5. User searches for the event **.**
6. User finds the event **.**
7. User may not be able to move on to the purchasing pos.

***b.** User selected expired/past events:

1. The user will be notified by an alert that the event does not exist or expire.
2. User can dismiss the alert and browse other events.

***c.** The user is logged in:

1. User may chose to browse events.
2. The events must not be expired.

3. User may select an event and come back from purchasing terminal, without actually purchasing any ticket.

***d**. No event exists:

1. In case no event exist, event page/window notifies the user that currently there is no available events in Charleston.

Special Requirements:

- The UI must be user friendly.
- The font and UI elements must follow Apple/Android interface guidelines.
- The event titles must be bold
- dates and description of the event must be smaller than the event title.

Technology and Data Variations list:

- The application must run on IOS and Android Platform.

VISION

Our vision is to create a ticking platform to be used campus wide! This platform will be used largely with performances on campus, but hopefully for more as well. Users will be able to buy tickets at associated events with the College of Charleston. The project will be built using React Native and JavaScript. The mobile app will be able to run on all mobile devices (Androids, iPhones, etc.). The estimated time of delivery of the final product will be April 2022.

GLOSSARY

Terms:	Definitions:
items	Product of service for sale
tickets	Electronic form of entry to specified show

SUPPLEMENTAL SPECS

Revision History:

Version:	Date:	Description:	Author:
Inception Draft	Sept 15, 2022	First Draft	Anil Chandler

Introduction
This document is the repository of CharlestonEvents requirements which are not shown in use cases

Functionality

General Use

Program should allow users to log in as well as browse shows, enter card information.

Logging

Program should be able to accurately log discrepancies found and report them to make the software maintenance process efficient.

Security

Program should be able to securely transfer sensitive data across servers. This includes data for important documents such as photo ID's, passports, credit card and debit card numbers.

Usability

Human Factors
Customers will be able to see
*Text should be easily visible from 1 meter
*Avoidance of colors which can be associated with color blindness

In a system which uses POS to transact tickets, it is important that the transaction is done quickly and efficiently.

Reliability

Recoverability
If there is an external issue with the system. There should be a safety net to complete the transaction from the local unit. In order to finalize a solution for this, we will do more planning here.

Performance
The users should be able to use the application without much delay while buying tickets. A recommended time frame for the average user to use the application is around 5 minutes. This is from the moment the application is opened to the purchasing and closing of the app is completed.

Supportability

Configurability
This application will have a fairly standard look for the user interface. However, if there is extra time for this project; a light/dark mode will be implemented for the user to configure to their liking.

Implementation Constraints
CharlestonEvents will be using React Native. By doing so, it should implement ease of development and make the generation of the final product feasible.

Free Open Source Components
We will be using a majority of React Native libraries done on our own side of the project. There might be some instances in which we need to check on certain aspects of implementation:

- Database framework and construction

Interfaces

Noteworthy Hardware and Interfaces

- Touch Screen Monitor

Software Interfaces

- Credit/Debit Card Verification
- Receipt Confirmations
- Signature Verification

Application-Specific Domain (Business) Rules
These rules will be followed by pricing of events through the College of Charleston with the sourcing coming from their websites. This means that the same pricing rules apply.

Legal Issues

If there are open sourced components being used, the creators of those ideas must not be restricted as copyright. The development of this software must follow all rules associated with this.
All tax rules must, by law, be applied during sales. These are usually stable and rarely change.

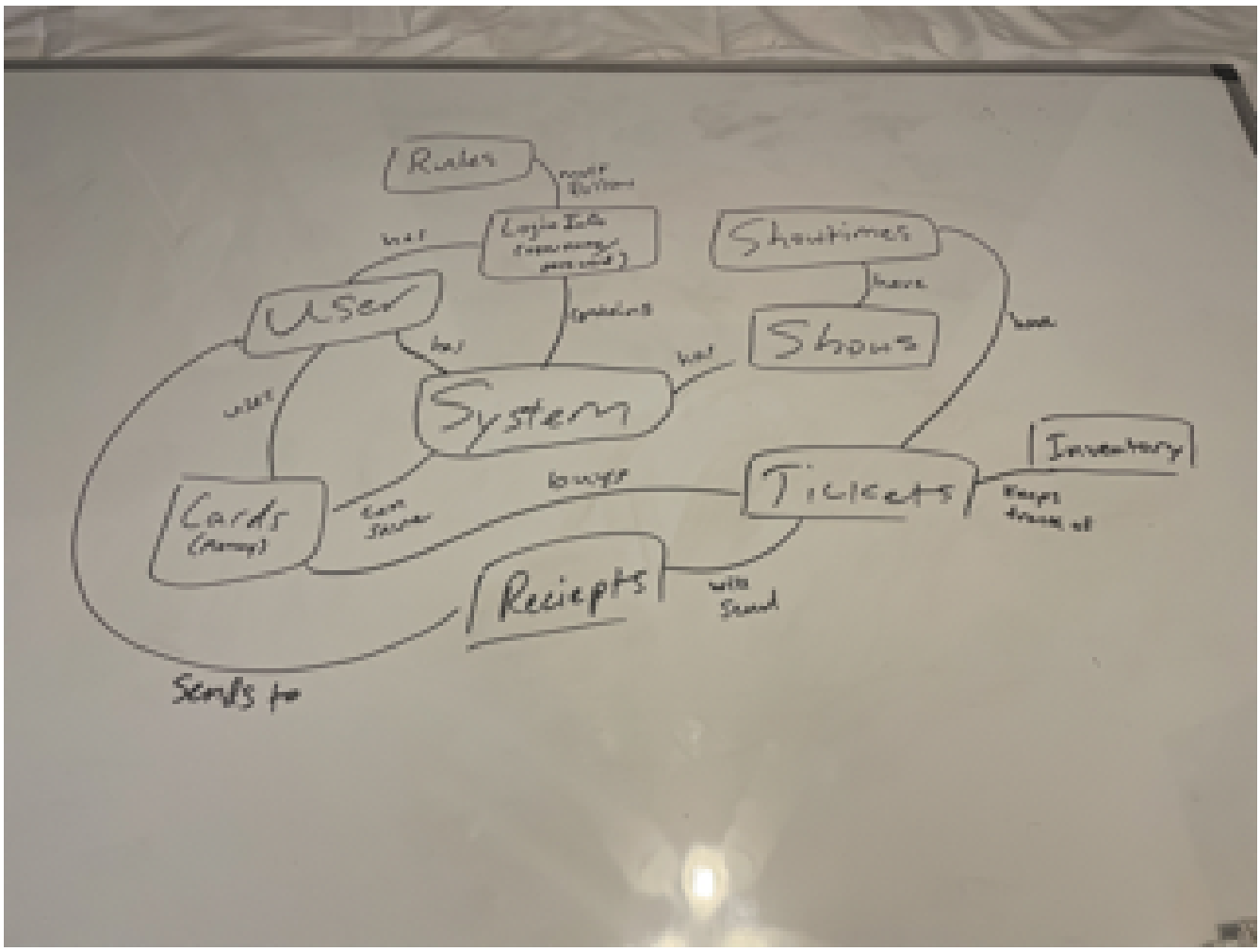
Information in Domains of Interest

Pricing
Products have an original price updated by the College of Charleston. Any changes to prices will be immediately reflected in the ticket price on the software. This includes markdowns and increases alike.

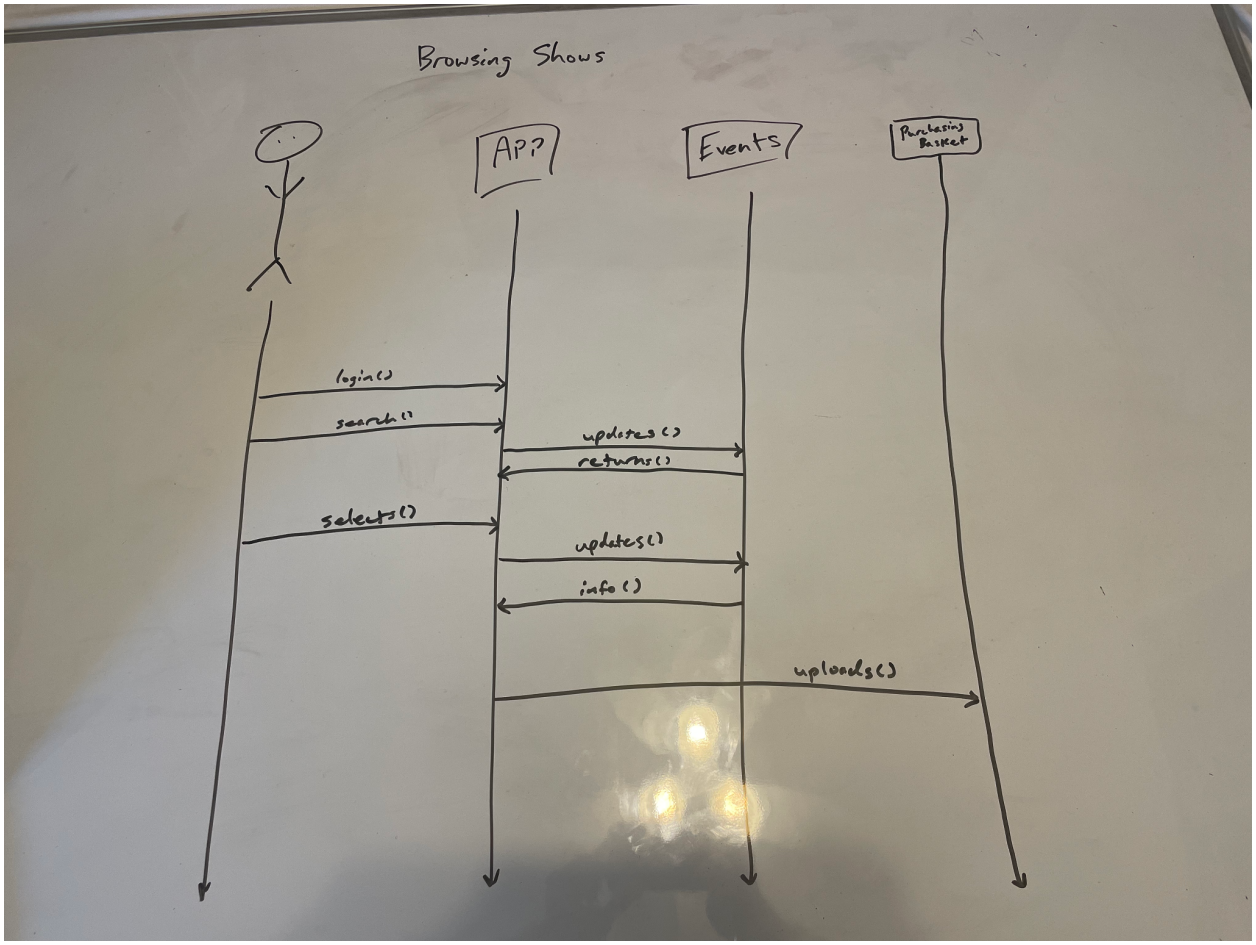
Credit and Debit Payment Handling
The buyer must be responsible for having sufficient funds in these accounts when purchasing tickets. It is the job of the seller to track receiving funds from buyers on a day to day basis to confirm accuracies and discrepancies.

Sales Tax
Sales tax calculations are able to change at the approval of governmental influence. Having a third-party application to calculate the sales tax might be more beneficial in the long run as these changes would be able to be caught faster and more accurately. Some items may be considered tax exempt depending on a certain buyer or certain recipient.

DOMAIN MODEL



Sequence System Diagram #1 - Browsing Shows



FOR BROWSING

Operation: OPENS

Cross References: NONE

Preconditions: none

Postconditions: FIRST VIEW PRESENTS

Operation: LOGIN

Cross References: USE CASE NOT DEFINED

Preconditions: NONE

Postconditions:

- **A SalesLineItem instance was created (instance creation).**
- **The App associated with the current user (association formed)**

Operation:Searching

Cross References: Use Cases: Browsing

Preconditions: logged In

Postconditions:

- **Event instance was created (instance creation)**
- **Display certain amount of events (attribute modification).**
- **Event was associated with a Event Details, based on itemID match (association formed).**
- **A Payment instance p was created (instance creation).**
- **p.amountTendered became amount (attribute modification). – p was associated with the current user formed).**
- **The current Sale was associated with the event organizer (association formed)**

Operation:PURCHASING

Cross References: Use Cases: PURCHASING

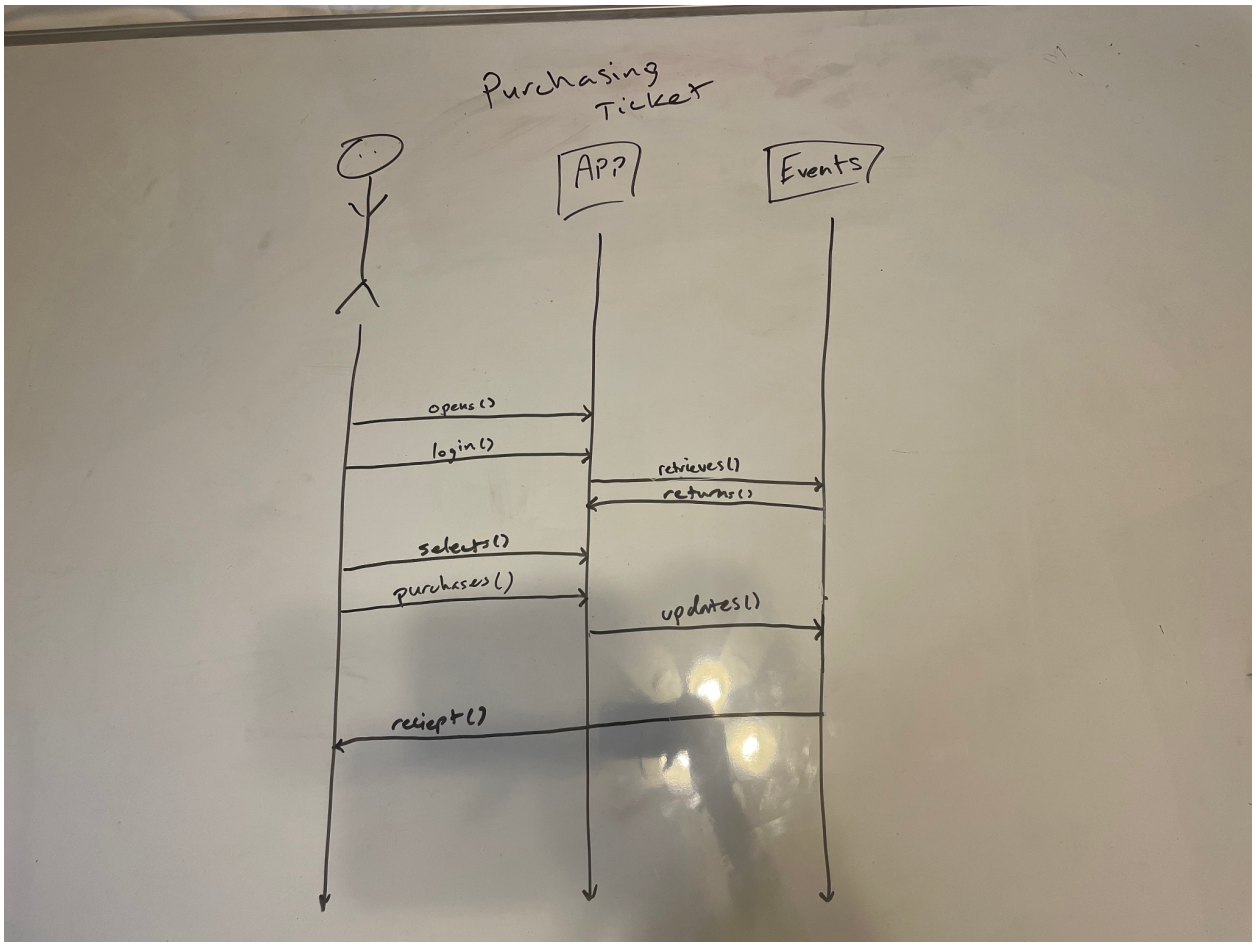
Preconditions: BROWSING EVENT

Postconditions:

- **A Payment instance p was created (instance creation).**

- **p.amountTendered became amount (attribute modification). – p was associated with the current user formed).**
- **The current Sale was associated with the event organizer (association formed)**

Sequence System Diagram #2 - Purchasing Tickets



Purchasing Tickets Operation Contract

Operation: open()

Cross reference: none

Preconditions: none

Postconditions: none

Operation: login()

Cross reference: Use case not defined

Preconditions: None

Postconditions:

- Login Instance “LGI” has been created for login operation.
- LGI has been associated with the User.

Operation: purchases()

Cross reference: Use case: Purchasing a ticket

Preconditions: User should be logged in

Postconditions:

- A payment instance “p” has been created
- p.TotalAmount became Total Amount

Operation: receipt()

Cross-reference: Use case has not been defined

Preconditions: User purchased a ticket

Postconditions:

- A receipt instance “rcp” has been created
- Rcp has been associated with the User

Operation: retrieves()

Cross-reference: User case has not been created

Preconditions: User must be logged in

Postconditions:

- An instance events has been created
- Events has been displayed in the app

Operation: updates()

Cross-reference: User case has not been defined

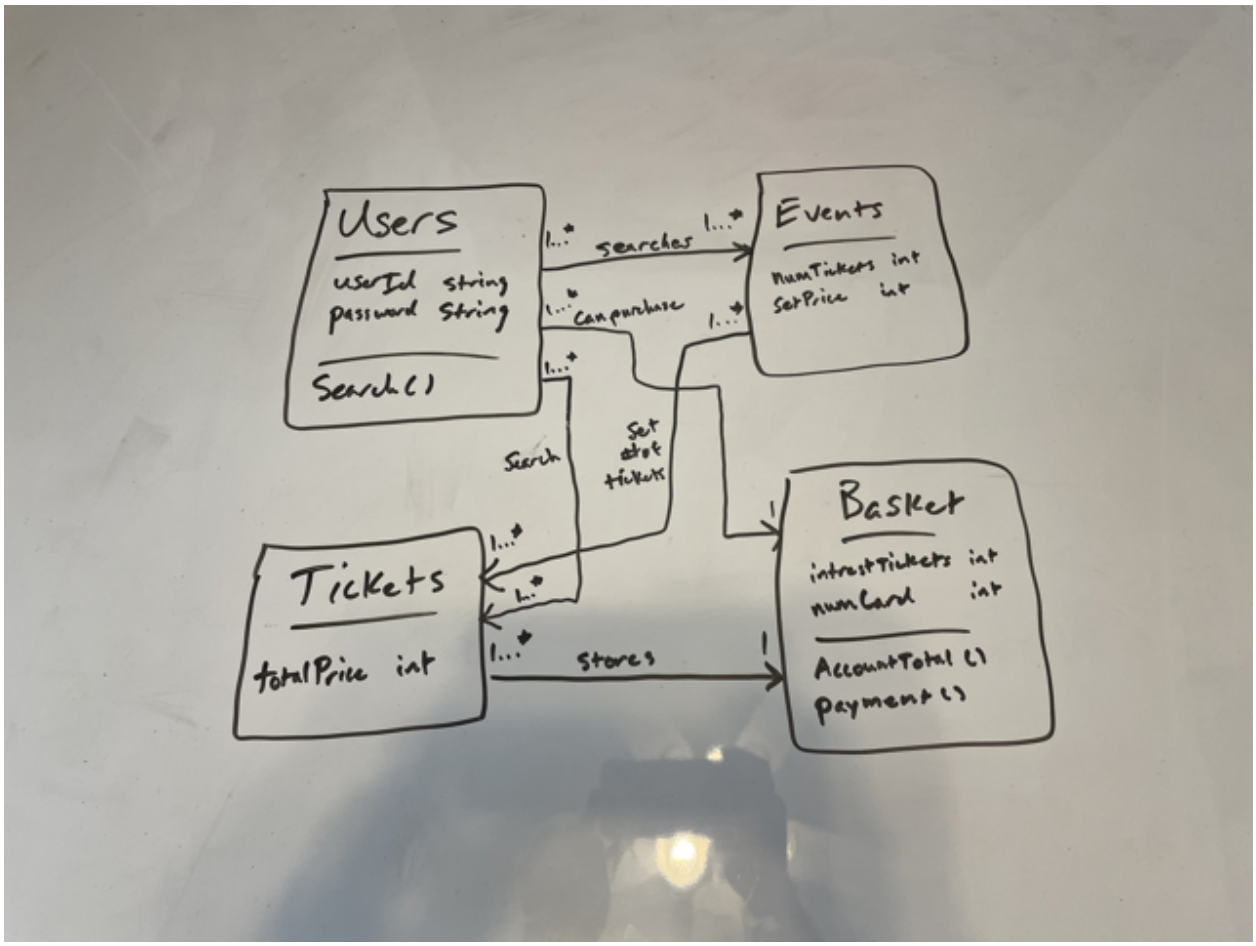
Preconditions:

- User must be logged in
- Events needs to have modifications

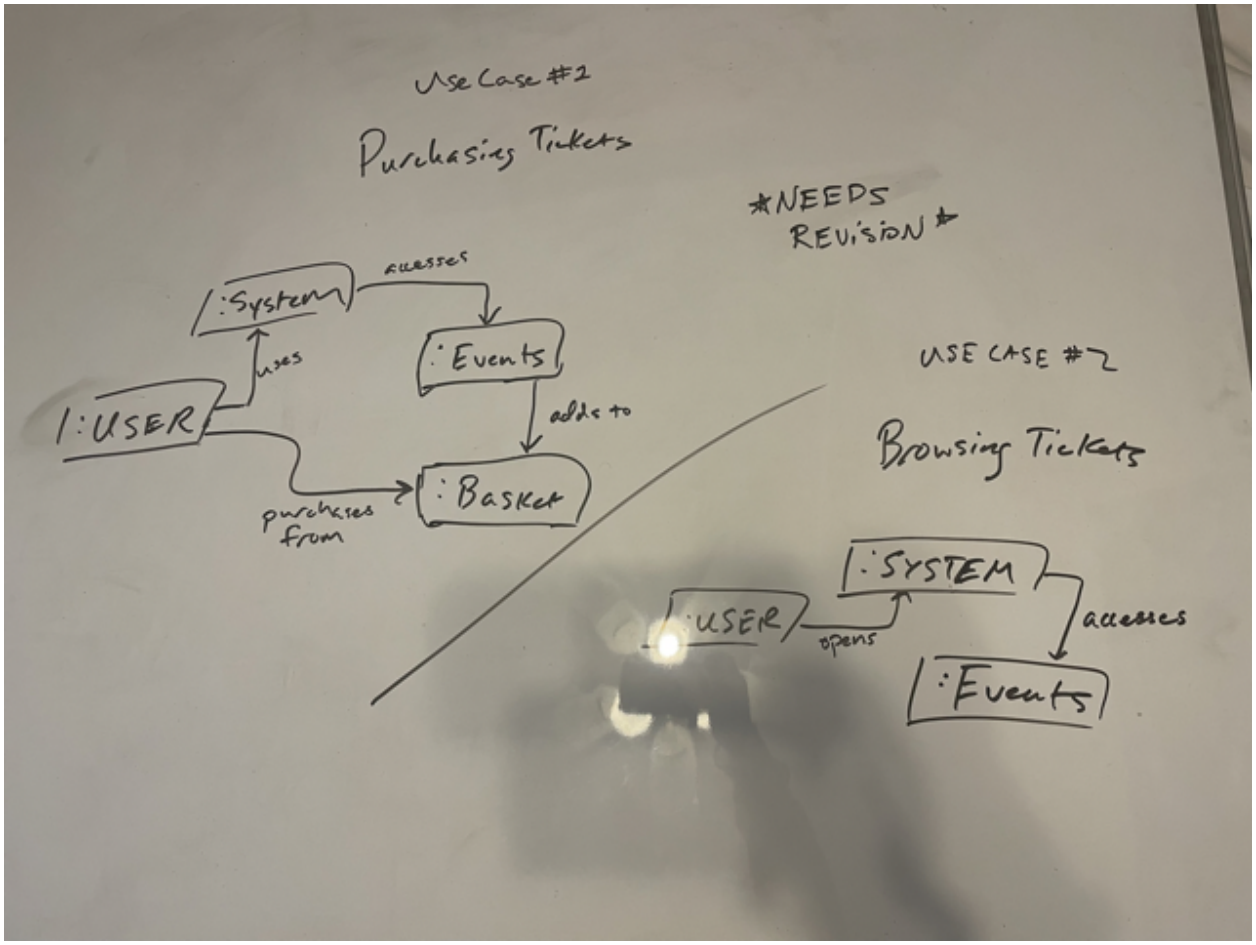
Postconditions:

- An update instance “upd” has been created
- upd.TotalAmtTicketsLeft became Tickets amount

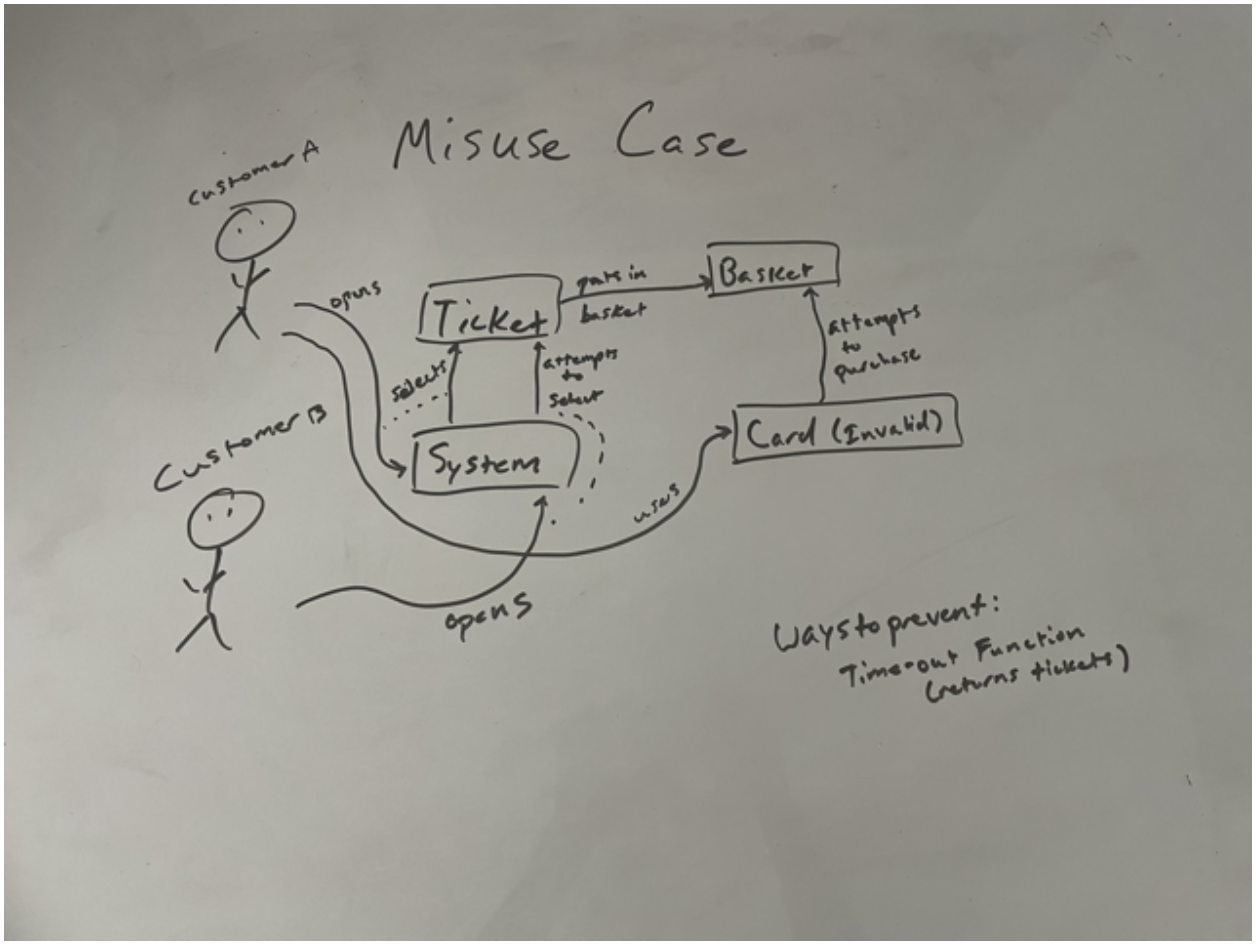
UML Class Diagram

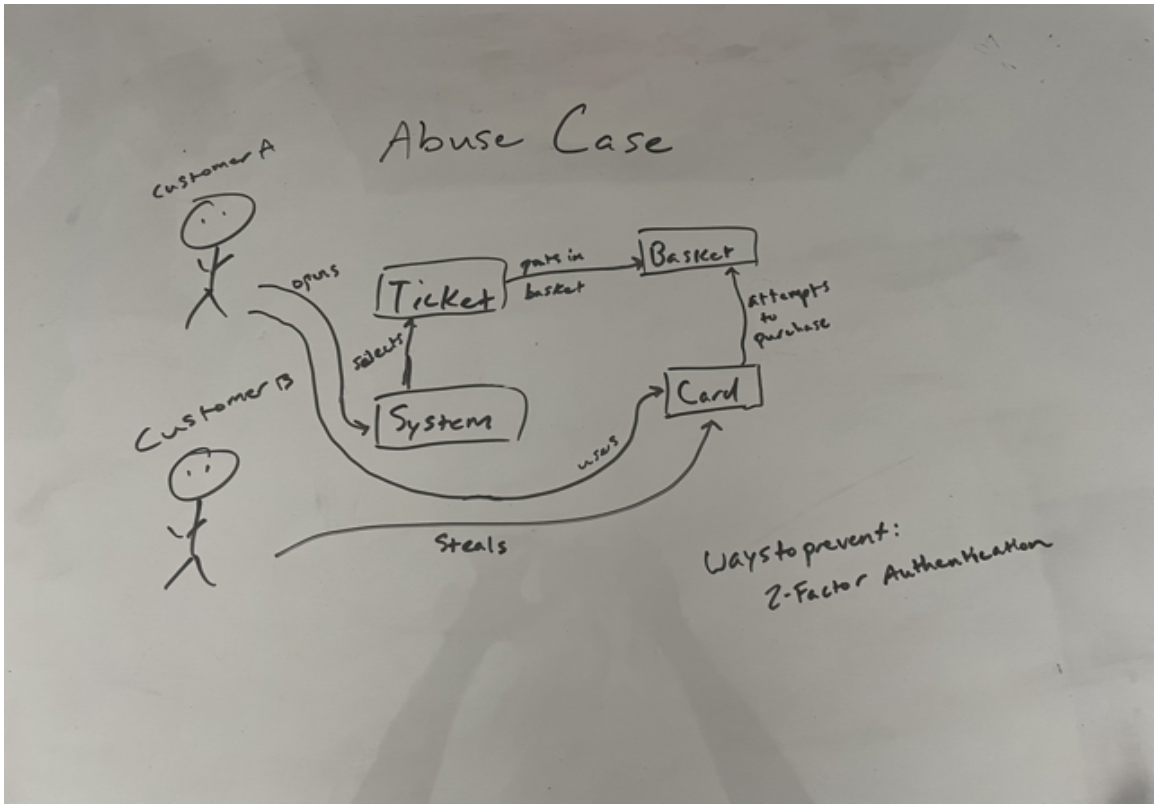


:System Object Application Layer - *Needs Revision*



Misuse Case - Holding onto Excess Tickets





Abuse Case - Stealing Card Info

UPDATE OPERATION CONTRACTS:

FOR BROWSING

Operation: OPENS

Cross References: NONE

Preconditions: none

Postconditions: FIRST VIEW PRESENTS

Operation: SignUp

Cross References: NONE

Preconditions: User Is not logged in

Postconditions:

- **SignUp instance “SU” has been created.**
- **“SU” has been associated with SignUp.**

Operation: LOGIN

Cross References: USE CASE NOT DEFINED

Preconditions: NONE

Postconditions:

- A SalesLineItem instance was created (instance creation).

- The App associated with the current user (association formed)

Operation: Verifies()

Cross Reference: none

Precondition: User tries to login

PostCondition:

- Two Step Verification, “TSV” instance has been created for TwoStepVerification operation.

- TSV.two step Verification has become the verification.

Operation:Searching

Cross References: Use Cases: Browsing

Preconditions: logged In

Postconditions:

- **Event instance was created (instance creation)**
- **Display certain amount of events (attribute modification).**
- **Event was associated with a Event Details, based on itemID match (association formed).**
- **A Payment instance p was created (instance creation).**
- **p.amountTendered became amount (attribute modification). – p was associated with the current user formed).**
- **The current Sale was associated with the event organizer (association formed)**

Operation:PURCHASING

Cross References: Use Cases: PURCHASING

Preconditions: BROWSING EVENT

Postconditions:

- **A Payment instance p was created (instance creation).**
- **p.amountTendered became amount (attribute modification). – p was associated with the current user formed).**
- **The current Sale was associated with the event organizer (association formed)**

Purchasing Tickets Operation Contract

Operation: open()

Cross reference: none

Preconditions: none

Postconditions: none

Operation: login()

Cross reference: Use case not defined

Preconditions: None

Postconditions:

- Login Instance “LGI” has been created for login operation.
- LGI has been associated with the User.

Operation: SpamCheck()

Cross Reference: none

Precondition: User has somany tickets in the basket

PostCondition:

- Spam check, “SC” instance has been created for Spam Check operation.
- SC.spamCheck is associated with spamCheck operation.

Operation: purchases()

Cross reference: Use case: Purchasing a ticket

Preconditions: User should be logged in

Postconditions:

- A payment instance “p” has been created

- p.TotalAmount became Total Amount

Operation: receipt()

Cross-reference: Use case has not been defined

Preconditions: User purchased a ticket

Postconditions:

- A receipt instance “rcp” has been created
- Rcp has been associated with the User

Operation: retrieves()

Cross-reference: User case has not been created

Preconditions: User must be logged in

Postconditions:

- An instance events has been created

- Events has been displayed in the app

Operation: updates()

Cross-reference: User case has not been defined

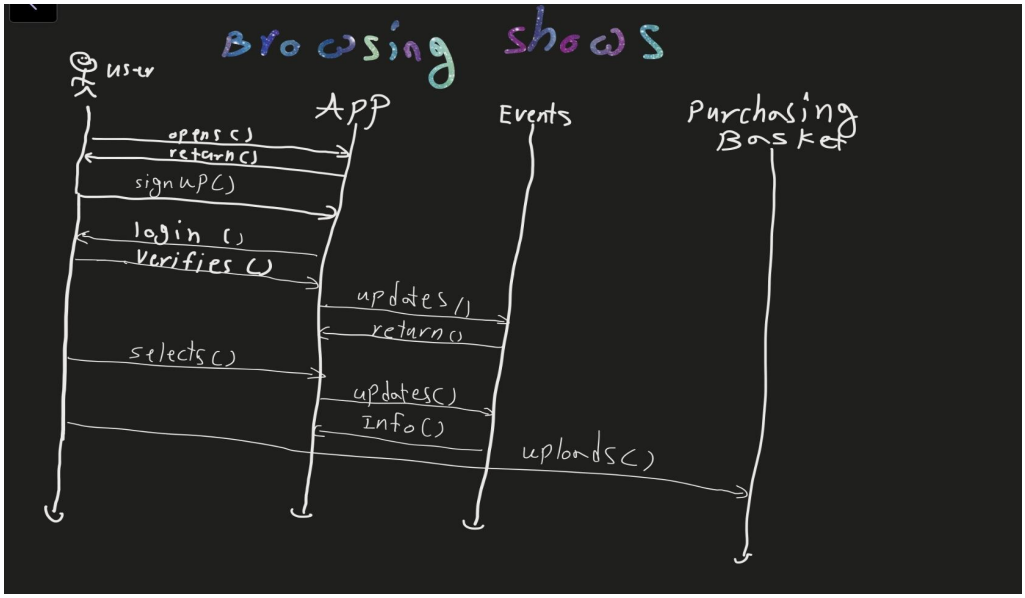
Preconditions:

- User must be logged in
- Events needs to have modifications

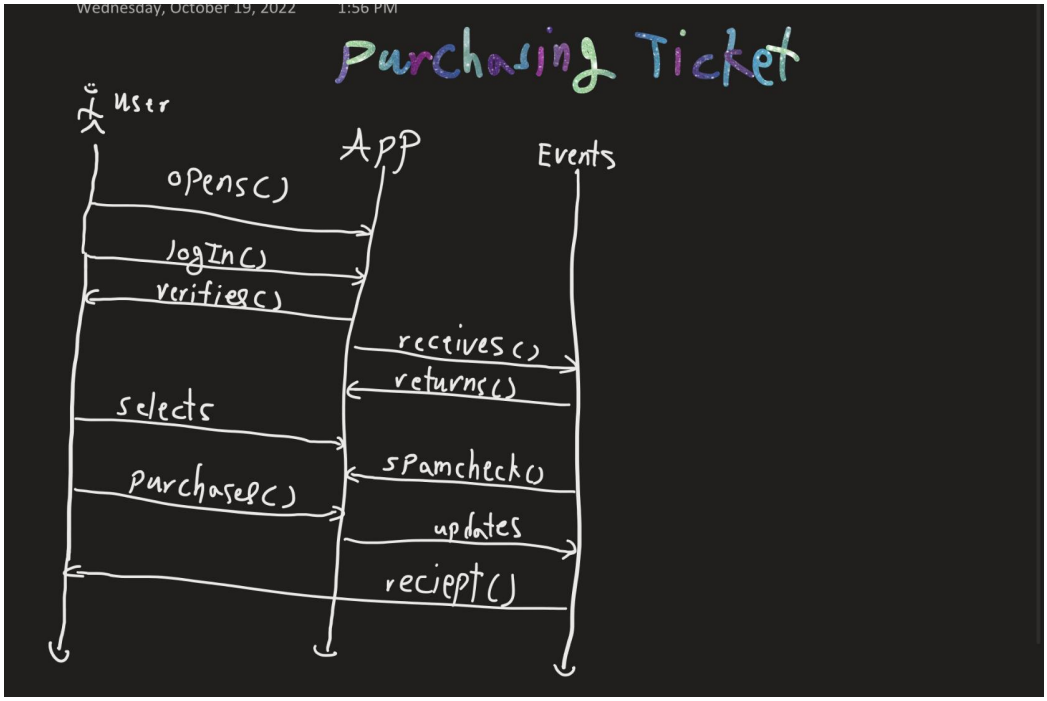
Postconditions:

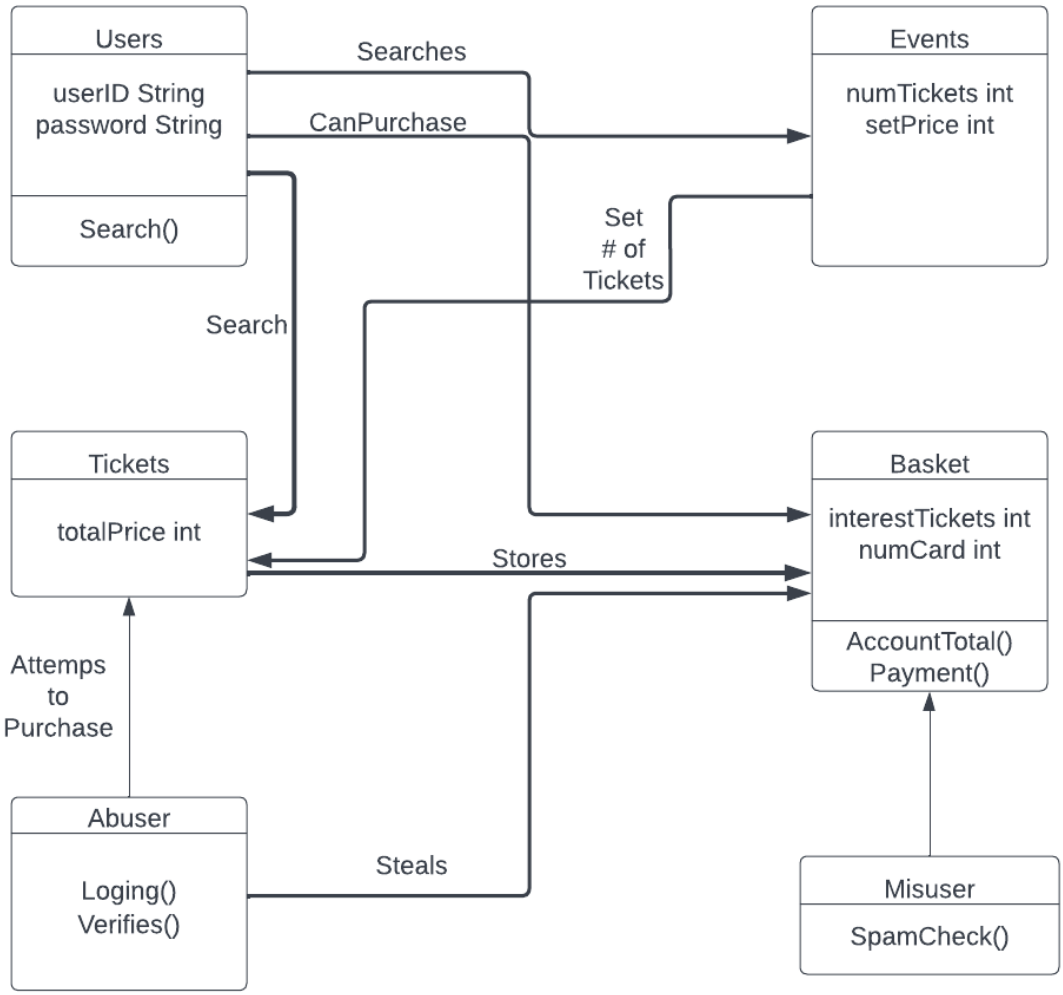
- An update instance “upd” has been created
- upd.TotalAmtTicketsLeft became Tickets amount

NEW SSD FOR PURCHASING AND BROWSING VIEW CONTROLLER



Wednesday, October 19, 2022 1:56 PM





OPERATION CONTRACTS WITH TECHNICAL ANALYSIS

FOR BROWSING

Operation: OPENS

Cross References: NONE

Preconditions: none

Postconditions: User has clicked on the application

Operation: SignUp

Cross References: NONE

Preconditions: User Is not logged in

Postconditions:

- **SignUp instance “SU” has been created.**
- **“SU” has been associated with SignUp.**
- **User is matched with a string Username and Password**

Operation: LOGIN

Cross References: USE CASE NOT DEFINED

Preconditions: NONE

Postconditions:

- A SalesLineItem instance was created (instance creation).
- The App associated with the current user (association formed)
- The App has the option to remember username and password for later use

Operation: Verifies()

Cross Reference: none

Precondition: User tries to login

PostCondition:

- Two Step Verification, “TSV” instance has been created for TwoStepVerification operation.
- TSV.two step Verification has become the verification.
- System matches Username / Password to Phone # is system
- System receives phone number code confirmation from user

Operation:Searching

Cross References: Use Cases: Browsing

Preconditions: logged In

Postconditions:

- **Event instance was created (instance creation)**
- **Display certain amount of events (attribute modification).**
- **Event was associated with a Event Details, based on itemID match (association formed).**
- **A Payment instance p was created (instance creation).**
- **p.amountTendered became amount (attribute modification). – p was associated with the current user formed).**
- **The current Sale was associated with the event organizer (association formed)**

For Purchasing Tickets

Operation: open()

Cross reference: none

Preconditions: none

Postconditions: none

Operation: login()

Cross reference: Use case not defined

Preconditions: None

Postconditions:

- Login Instance “LGI” has been created for login operation.
- LGI has been associated with the User.

Operation: SpamCheck()

Cross Reference: none

Precondition: User has somany tickets in the basket

PostCondition:

- Spam check, “SC” instance has been created for Spam Check operation.
- SC.spamCheck is associated with spamCheck operation.

Operation: Purchasing()

Cross References: Use Cases: PURCHASING

Preconditions: BROWSING EVENT

Postconditions:

- **A Payment instance p was created (instance creation).**
- **p.amountTendered became amount (attribute modification). – p was associated with the current user formed).**
- **The current Sale was associated with the event organizer (association formed)**

Operation: receipt()

Cross-reference: Use case has not been defined

Preconditions: User purchased a ticket

Postconditions:

- A receipt instance “rcp” has been created
- Rcp has been associated with the User
- System sends an already generated email from admin to user using Email from purchase

Operation: retrieves()

Cross-reference: User case has not been created

Preconditions: User must be logged in

Postconditions:

- An instance events has been created
- Events has been displayed in the app

Operation: updates()

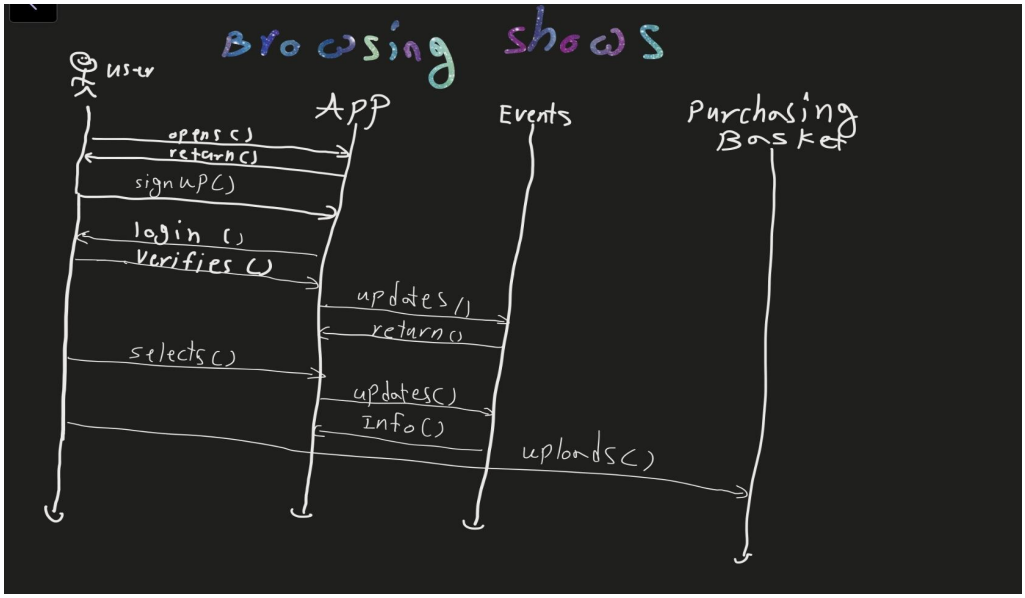
Cross-reference: User case has not been defined

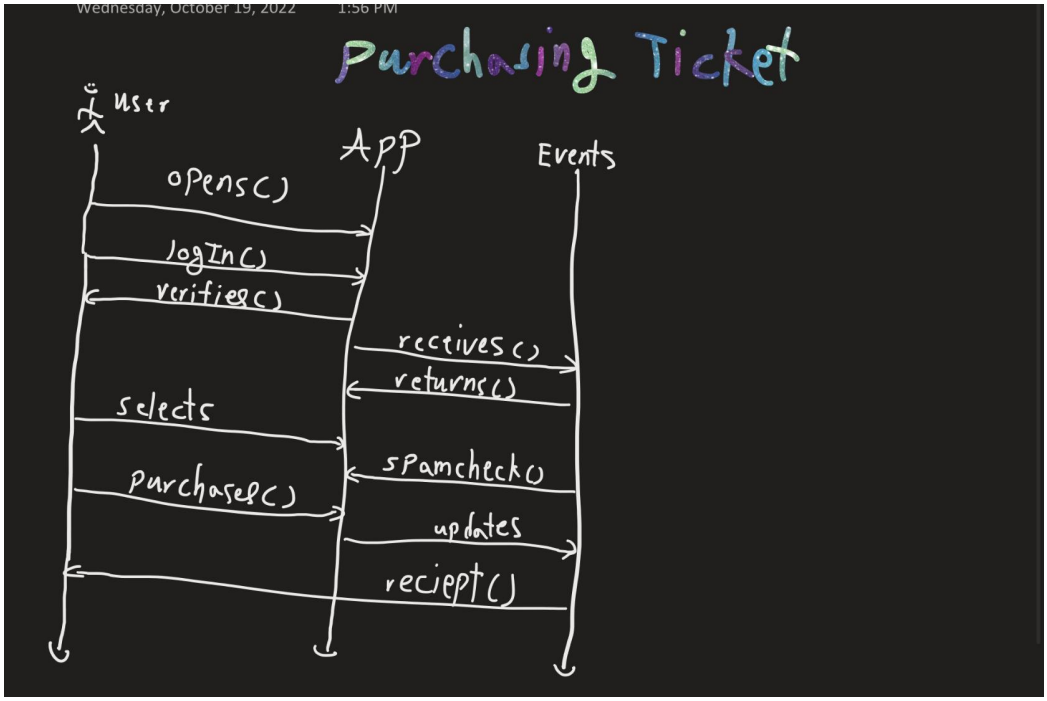
Preconditions:

- User must be logged in
- Events needs to have modifications

Postconditions:

- An update instance “upd” has been created
- upd.TotalAmtTicketsLeft became Tickets amount
- Ticket counter is then subtracted from the total by number of tickets bought
    - This number is shown when others are purchasing tickets to see how much is left





LEAST COUPLED JAVA CLASS

public class Tickets {

```
        private int price;
        private boolean owned;
        // Setters //
         private void setPrice(int price) {
         this.price = price;
        }
        private void setOwnership(boolean owned) {
        this.owned = owned;
         }
        // Getters //
        public int getPrice() {
        return this.price;
        }
        private in getOwnership(){
        return this.ownership;
        }
}
```

JAVA CLASS IMPLEMENTATION

```java
public class Basket {



   private int numberOfTickets;

   private int numCard;



   public static int accountTotal() {

       System.out.print("calculates Total: ");
```

```
    return -1; //temporary
}

private void payment(){
    System.out.print("receive payment: ");
    //receive payment
}

// Getters //
public void setNumOfTickets(int nt ) {
    this.numberOfTickets = nt;
}

  private void setnumCard(int nc ) {
    this.nc = nc;
}

// Setters //

private int getNumOfTickets(int numberOfTickets) {
   return this.numberOfTickets = numCard;

}
private int gettnumCard(int numCard) {
```

```
        return this.numCard = numCard;

    }




    public static <string> void main(String args) {

        //object

        //create objects form other calsses

        MusUser Misuser = new MusUser();

        //object from Abuser class

        AbUser Abuser = new AbUser();

        //object from Tickets class

        Tickets Tickets = new Tickets();

        //object from Users class

        Users Users = new Users();







        System.out.print("This is the Basket Class!");

    }

}
```

```java
public class Events {

   private int numTickets;

   private int price;



  public void setNumTickets(int ticketsNumber){ //set number of tickets available

       this.numTickets = ticketsNumber;

   }



   public void setPrice(int amount){ //set price of tickets available

       this.price = amount;

   }



   public int getPrice(){ //get the price of the ticket

       return this.price;

   }



   public int getNumTickets(){ //get the number of tickets available

       return this.numTickets;

   }

}
```

EVENTS TEST

```
public class Events {

   private int numTickets;

   private double price;

   public String eventName;



   public void setEventName(String name){

       this.eventName = name;

   }

   public void setNumTickets(int ticketsNumber){
```

```
       if(ticketsNumber < 0){

           throw new IllegalArgumentException("There can not be negative amount of
tickets");

       }

       this.numTickets = ticketsNumber;
```

```
       //ticketsNumber should not negative.

   }

    public void setPrice(double amount){

       if (amount < 0){

           throw new IllegalArgumentException("Price should be at least $0.00");

       }

           this.price = amount;
```

```
       //Price has to be above $0.

   }

    public double getPrice(){

       return this.price;
```

```
   }

    public int getNumTickets(){

        return this.numTickets;

   }
```

```
   public String getEventName(){

        return this.eventName;

   }
```

```
   public String toString() {

        return "Event: " + this.eventName + "\nTickets available: " + this.numTickets +
"\nPrice: $" + this.price;

   }
```

```
public static void main(String[] args){
```

```
    //TEST 1 -- Positive Price
```

```
    Events event3 = new Events();
```

```
    event3.setEventName("Computer Science Career Fair");

    event3.setNumTickets(50);

    event3.setPrice(25);
```

```
    System.out.println(event3.toString());
```

```
    // TEST 2 -- Positive Amount of Tickets
```

```
    Events event4 = new Events();
```

```
    event4.setEventName("WIC: Github Workshop");
    event4.setNumTickets(15);
    event4.setPrice(35);

    System.out.println(event4.toString());

   //TEST 3 -- Negative Price
       Events event1 = new Events();

       event1.setEventName("Computer Science Career Fair");
       event1.setNumTickets(30);
       event1.setPrice(-5.35);


   //TEST 4 -- Negative Amount of Tickets

       Events event2 = new Events();
       event2.setEventName("WIC: Github Workshop");
       event2.setNumTickets(-4);
       event2.setPrice(0);


}


}
```

Implemented Java Code with JUnit Testing

Basket.java

```
import org.junit.*;

public class UserTest {

   static User aUser = new User();

   public void setupUser() {
       aUser.setuserID("New_Username");
       aUser.setPassword("New_Password");
   }

   public void testGetUserID(){
           // Expected Variable, Actual Variable, Compare
       String expected = "New_Username";
       String actual = aUser.getuserID();
       assertEquals("Testing Get UserID", expected, actual);
   }


   public void testGetPassword() {
           // Expected Variable, Actual Variable, Compare
       String expected = "New_Password";
       String actual = aUser.getPassword();
       assertEquals("Testing Get Password", expected, actual);
   }
                                              }
```

BasketTest.java (JUnit)

```
import static org.junit.Assert.assertEquals;

import org.junit.*;
import org.junit.runners.Parameterized.BeforeParam;

public class BasketTest {
   static Basket aBasket = new Basket();

   @BeforeClass
   public static void setUpClass(){
```

```
        aBasket.setNumOfTickets(5);
        aBasket.settnumCard("123456789");
    }

    @Test
    public void testGetNumCard(){
        String expected = "123456789";
        String actual = aBasket.getnumCard();

        assertEquals(expected, actual);
    }

    @Test
    public void testGetNumOfTickets(){
        int expected = 5;
        int actual = aBasket.getNumOfTickets();

        assertEquals(expected, actual);
    }
}
```

Events.java

```
public class Events {
    private int numTickets;
    private double price;
    public String eventName;

    public void setEventName(String name){
        this.eventName = name;
    }
    public void setNumTickets(int ticketsNumber){

        if(ticketsNumber < 0){
            throw new IllegalArgumentException("There can not be negative amount of
tickets");
        }
        this.numTickets = ticketsNumber;

        //ticketsNumber should not negative.
```

```
    }
     public void setPrice(double amount){
        if (amount < 0){
            throw new IllegalArgumentException("Price should be at least $0.00");
        }
            this.price = amount;

        //Price has to be above $0.
    }
     public double getPrice(){
        return this.price;
    }
     public int getNumTickets(){
        return this.numTickets;
    }

    public String getEventName(){
        return this.eventName;
    }

    public String toString() {
        return "Event: " + this.eventName + "\nTickets available: " + this.numTickets +
"\nPrice: $" + this.price;
    }

public static void main(String[] args){

     //TEST 1 -- Positive Price

     Events event3 = new Events();

     event3.setEventName("Computer Science Career Fair");
     event3.setNumTickets(50);
     event3.setPrice(25);

     System.out.println(event3.toString());

     // TEST 2 -- Positive Amount of Tickets

     Events event4 = new Events();

     event4.setEventName("WIC: Github Workshop");
```

```
     event4.setNumTickets(15);
     event4.setPrice(35);

     System.out.println(event4.toString());

    //TEST 3 -- Negative Price
        Events event1 = new Events();

        event1.setEventName("Computer Science Career Fair");
        event1.setNumTickets(30);
        event1.setPrice(-5.35);


    //TEST 4 -- Negative Amount of Tickets

        Events event2 = new Events();
        event2.setEventName("WIC: Github Workshop");
        event2.setNumTickets(-4);
        event2.setPrice(0);

}
}
```

EventsTest.java (JUnit)

```
import static org.junit.Assert.assertEquals;

import org.junit.*;
public class EventsTest {
   static Events anEvent = new Events();

   @BeforeClass
   public static void setUpClass(){
        anEvent.setEventName("WIC");
        anEvent.setNumTickets(5);
        anEvent.setPrice(53.45);
   }

   @Test
   public void testGetPrice(){
      double expected = 53.45;
      double actual = anEvent.getPrice();
```

```
        assertEquals(expected, actual, 0);
    }

    @Test
    public void testGetNumTickets(){
        int expected = 5;
        int actual = anEvent.getNumTickets();

        assertEquals(expected, actual);
    }

    @Test
    public void testGetEventName(){
        String expected = "WIC";
        String actual = anEvent.getEventName();

        assertEquals(expected, actual);

    }
}
```

Tickets.java

```
public class Tickets {

    private double price;
    private int number;

    public Tickets (){

        double price = 0;
        int Number = 0;

    }

    public Tickets (double ticketPrince , int ticketNumber ){

        double price = ticketPrince;
        int tNumber = ticketNumber;

    }
```

```
    // Setters //
    public void setPrice(double price) {
        this.price = price;
    }
     //setter
    public void setNumber(int number) {
        this.number = number;
    }

    //getter method
    public double getPrice() {
        return this.price;
    }
     //getter method
    public int getNumber() {
        return this.number;
    }

    //toStringMethod
    public String toString(){
        return ("This ticket Object Name"  + this.number + " Ticket Price: " +
this.price);
    }

    //

    public static boolean testCase1(Tickets ticket) {
        if (ticket.getPrice() <= 10) {
            return false;
        }
        return true;
    }

    public static boolean testCase2(Tickets numOfTickets){
        if(numOfTickets.getNumber() == 0){
            return false;
        }
        return true;
    }

    public static void main(String[] args) {
```

```
        Tickets a = new Tickets();
        a.setPrice(11);
        a.setNumber(0);
        System.out.println(a.toString());
        Tickets b = new Tickets();
        b.setPrice(10);
        b.setNumber(2);
        b.toString();
        Tickets c = new Tickets();
        c.setPrice(10);
        c.setNumber(2);
        c.toString();
        System.out.println(testCase1(a));
        System.out.println(testCase2(a));


    }
}
```

TicketTest.java (JUnit)

```
import static org.junit.Assert.assertEquals;

import org.junit.*;

public class TicketsTest {
   static Tickets aTicket = new Tickets();

   @BeforeClass
   public static void setUpClass(){
      aTicket.setPrice(25.00);
      aTicket.setNumber(50);
   }

   @Test
   public void testGetPrice(){
      double expected = 25.00;
      double actual = aTicket.getPrice();

      assertEquals(expected, actual, 0);
   }
```

```
    @Test
    public void testGetNumber(){
        int expected = 50;
        int actual = aTicket.getNumber();

        assertEquals(expected, actual);
    }
}
```

User.java

```
public class User {
    private String password;
    private String userID;

    public User() {
        password = null;
        userID = null;
    }

    public User(String a, String b) {
        userID = a;
        password = b;
    }

    // Getters //
    void setuserID(String uID) {
        this.userID = uID;
    }

    String getuserID() {
        return this.userID;
    }

    // Setters //

    void setPassword(String pass) {
        this.password = pass;

    }
    String getPassword() {
```

```
        return this.password;
    }

    public String toString() {
        return "=== User Information === \nUserID: "+this.userID +" \nPassword: "+
this.password + "\n";
    }

    public static void main(String[] args) {
        // TEST ONE //
        User a = new User();
        System.out.print(a.toString());
        // TEST TWO //
        User b = new User("null", "pass");
        System.out.print(b.toString());
        // TEST THREE //
        User c = new User("Clear1", "");
        System.out.print(c.toString());

        // EXTRA TESTS - GETTERS/SETTERS //

    }

}
```

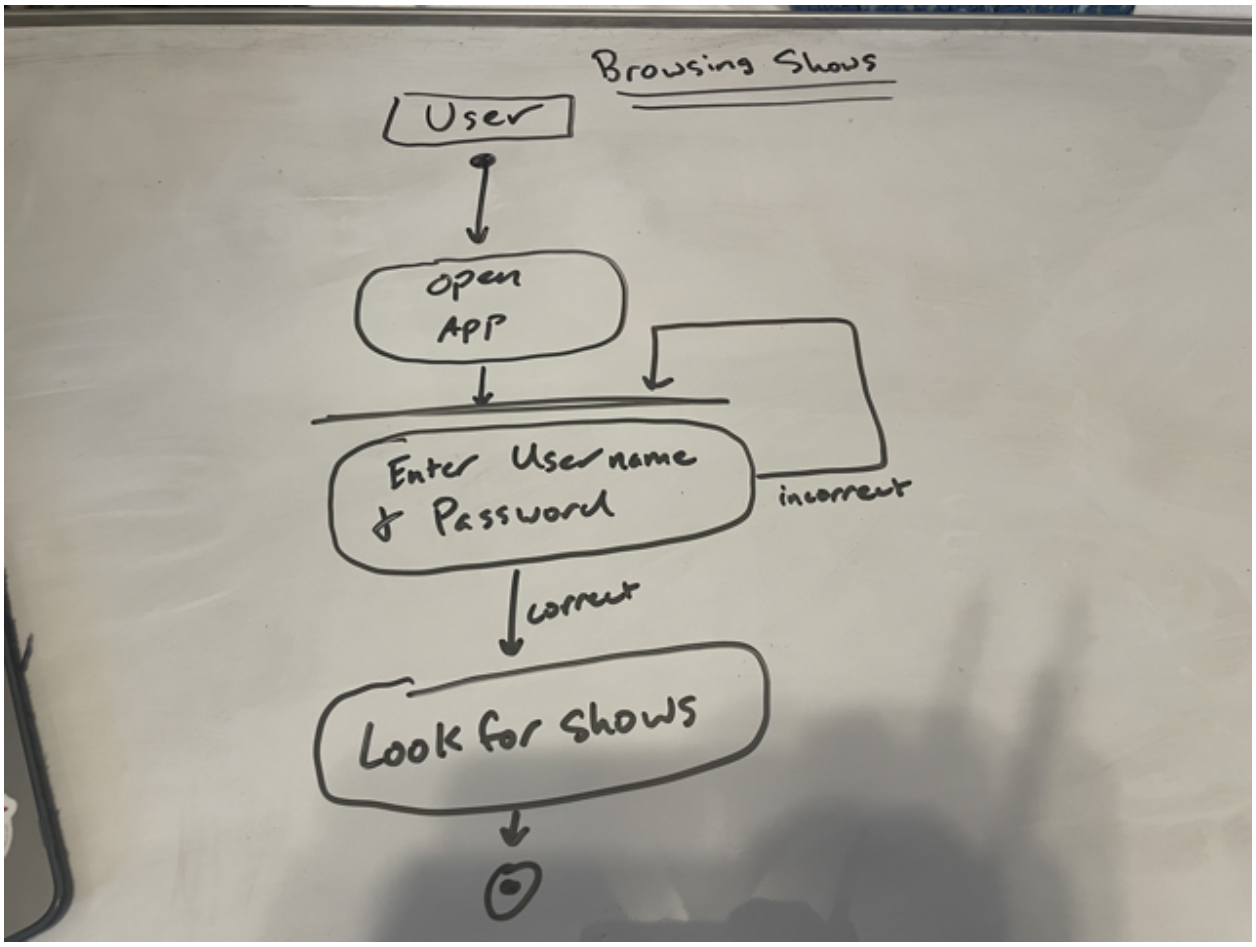
UserTest.java (JUnit)

```
import static org.junit.Assert.assertEquals;

import org.junit.*;

public class UserTest {

    static User aUser = new User();

    @BeforeClass
    public static void setupUser() {
        aUser.setuserID("New_Username");
        aUser.setPassword("New_Password");
    }

    @Test
```

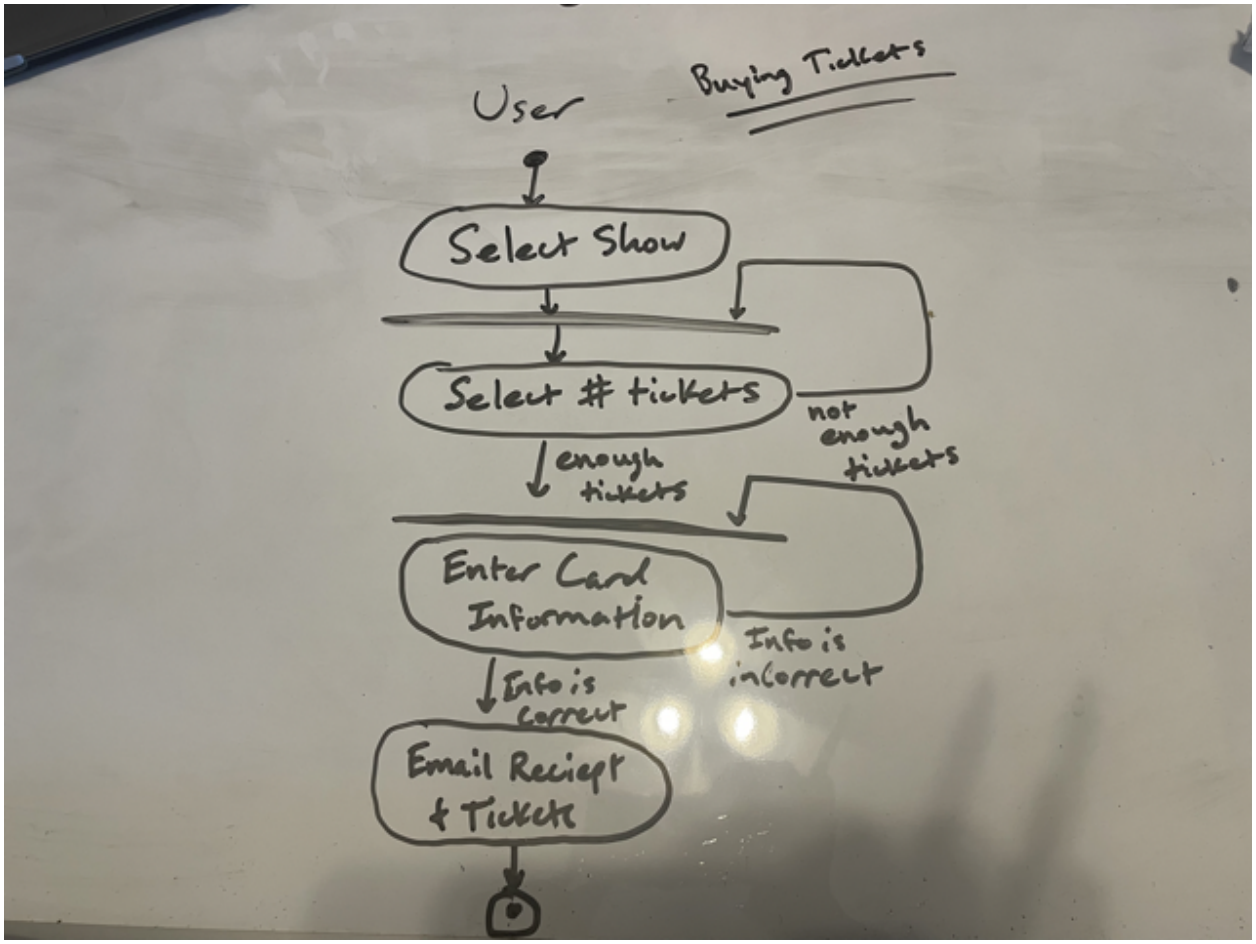
```
    public void testGetUserID(){
        String expected = "New_Username";
        String actual = aUser.getuserID();

        assertEquals(expected, actual);
    }

    @Test
    public void testGetPassword() {
            // Expected Variable, Actual Variable, Compare
        String expected = "New_Password";
        String actual = aUser.getPassword();

        assertEquals(expected, actual);
    }

}
```

Activity Diagrams

Browsing Shows



Buying Tickets


Buying Tickets
User
Select Show
Select # tickets
not enough tickets
enough tickets
Enter Card Information
Info is incorrect
Info is correct
Email Reciept & Tickets