Amusement Park Scheduler
System Design

Version: 0.3.8 – Release March 10th, 2005

Created by: Allen Ma
Edited by: Gerard Mainardi, Jesse Rohland, Daniel Smith, Sharon Traineanu
<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Owner</th>
<th>Reviewer</th>
<th>Sections Affected/Changes Made</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1.1</td>
<td>2/11/05</td>
<td>Gerard</td>
<td>Dan</td>
<td>Added VERY preliminary DB section</td>
</tr>
<tr>
<td>0.2.1</td>
<td>2/18/05</td>
<td>Gerard</td>
<td>Jesse</td>
<td>Added more detailed DB diagrams and images</td>
</tr>
<tr>
<td>0.2.2</td>
<td>2/18/05</td>
<td>Sharon</td>
<td>Jesse</td>
<td>Added Class Diagrams</td>
</tr>
<tr>
<td>0.3.1</td>
<td>3/1/05</td>
<td>Jesse</td>
<td>Dan</td>
<td>Updated class diagrams, inserted class descriptions, inserted class relationship diagram</td>
</tr>
<tr>
<td>0.3.2</td>
<td>3/1/05</td>
<td>Dan</td>
<td>Gerard</td>
<td>Added framework description &amp; XML API formats</td>
</tr>
<tr>
<td>0.3.3</td>
<td>3/7/05</td>
<td>Gerard</td>
<td>Sharon</td>
<td>Overhauled the database section with descriptions</td>
</tr>
<tr>
<td>0.3.4</td>
<td>3/8/05</td>
<td>Sharon</td>
<td></td>
<td>Pre and Post conditions</td>
</tr>
<tr>
<td>0.3.5</td>
<td>3/8/05</td>
<td>Dan</td>
<td></td>
<td>Reformating and updating of some text</td>
</tr>
<tr>
<td>0.3.6</td>
<td>3/8/05</td>
<td>Jesse</td>
<td>Sharon</td>
<td>Added sequence diagrams and large database diagram</td>
</tr>
<tr>
<td>0.3.7</td>
<td>3/9/05</td>
<td>Sharon</td>
<td></td>
<td>Added Class Relationship Diagram, rearranged several sections</td>
</tr>
<tr>
<td>0.3.7</td>
<td>3/10/05</td>
<td>Sharon</td>
<td></td>
<td>Coding Standards and Implementation Plan</td>
</tr>
<tr>
<td>0.3.7</td>
<td>3/10/05</td>
<td>Sharon</td>
<td>Dan</td>
<td>Added Glossary</td>
</tr>
<tr>
<td>0.3.8</td>
<td>3/10/05</td>
<td>Jesse</td>
<td>Dan</td>
<td>Inserted processor descriptions</td>
</tr>
</tbody>
</table>
Table of Contents

1 Introduction ................................................................................................................................. 4
   1.1 Abstract ............................................................................................................................... 4
   1.2 Purpose ............................................................................................................................... 4
   1.3 Document Scope ................................................................................................................ 4

2 General Architecture ............................................................................................................... 5
   2.1 System Overview Diagram ................................................................................................. 5
   2.2 APS Framework .................................................................................................................. 5
   2.3 APS Processors .................................................................................................................. 7
   2.4 Class Relationship Diagram ............................................................................................... 8
   2.5 Namespace: APS .................................................................................................................. 8
   2.6 Namespace: APS.Framework ............................................................................................... 10
   2.7 Namespace: APS.Framework.Modules ................................................................................... 11
   2.8 Namespace: APS.Framework.CommonFunction ................................................................. 19
   2.9 Sequence Diagrams .......................................................................................................... 21

3 User Interface .......................................................................................................................... 25
   3.1 General Overview .............................................................................................................. 25
   3.2 Web Interface ..................................................................................................................... 25
   3.3 In-Park Kiosk Interface ...................................................................................................... 27
   3.4 Mobile Device – Text Interface ......................................................................................... 28
   3.5 Mobile Device – PDA Interface ......................................................................................... 29
   3.6 Purchaser Interface ............................................................................................................ 30

4 Data Collection ........................................................................................................................ 33
   4.1 General Overview .............................................................................................................. 33
   4.2 Database Overview Diagram ............................................................................................. 34
   4.3 Table Descriptions ............................................................................................................ 35

5 Coding Standards ..................................................................................................................... 43
   5.1 Naming Conventions and Style ......................................................................................... 43
   5.2 Coding Practices ................................................................................................................ 44
   5.3 .NET Framework Specific Guidelines .............................................................................. 45

6 Implementation Plan ............................................................................................................... 46
   6.1 Implementation Steps ........................................................................................................ 46
   6.2 Deployment ......................................................................................................................... 47

7 Appendix ................................................................................................................................. 48
1 Introduction

1.1 Abstract
Amusement Parks have recently begun implementing automated ticket systems to increase overall guest satisfaction by reducing time spent waiting in ride queues, and to track visitor traffic and riding patterns. We propose a system that collects a larger variety of park data and encourages increasing QuickTicket usage, resulting in more efficient park flow in addition to offering guests easy to use interfaces to improve their park experience. Additionally, APS encourages return visitors by providing guests with a more enjoyable visit.

1.2 Purpose
This document is intended to give an overview of the system architecture and design of the APS system. It is a complete document for basic system functionality and structure. It also describes system dependencies, relations, and inheritance hierarchies. This document will serve as the basis for implementation of the Amusement Park Scheduler system.

1.3 Document Scope
This document provides a complete design for the Amusement Park Scheduler system and specifies how the functional and non-function requirements as detailed in the Amusement Park Schedule Requirements Specification document will be satisfied.

All software architecture will be included in this document including that of packages, classes, and functions. Additionally, a prototype and state diagram of the user interfaces is included.

Note: This document does not fully describe a functional implementation of the APS system. Certain assumptions are made which, if proved wrong, could change design decisions during the implementation phase of the project.
2 General Architecture

2.1 System Overview Diagram
Below is a diagram depicting the general APS System design, including each class as described in sections 2.4 through 2.7, and the APS Framework as described in section 2.2.

2.2 APS Framework
The APS Framework is the backend portion of the APS System, holding all logic and system functionality. The APS Framework Service consists of a Windows NT Service that contains a series of system Modules (outlined in the Class section of this document), accessible to external interfaces through Microsoft's .NET Remoting. The Framework Service will host a solitary .NET Remote class with a single method, execute, which accepts an API request and returns a response in the XML formats specified in sections 2.2.1 and 2.2.2.

The Framework accepts each API request, instantiates the given module, and calls the requested static method with the provided parameters. The requested function's return value is then converted into an appropriate XML format and returned to the calling application. If an error occurred during execution of the request, an error code (outlined in section 2.2.4) will be returned along with an
error message. Additionally, the parameter types supported by the API are outlined in section 2.2.3.

All APS.Framework.Modules classes contain only static methods which perform any required logic, computations, or data retrieval or storage within the database or caching objects. These classes do not load or store data internally, as their instance is discarded upon completion of the API request; each API request is treated independently of the next, as these classes are not persistent.

2.2.1 API Request XML

```xml
<APSapi>
  <request id="request_id">
    <module>
      <!-- module being accessed -->
    </module>
    <function>
      <!-- function to be called within Module -->
    </function>
    <parameters>
      <param type="param_type"></param>
    </parameters>
  </request>
</APSapi>
```

2.2.2 API Response XML

```xml
<APSapi>
  <response id="request_id">
    <error id="error_code">
      <!-- error message-->
    </error>
    <return type="param_type">
      <!-- return value in XML format-->
    </return>
  </response>
</APSapi>
```

2.2.3 Supported Parameter Types

- Integer
- Float
- Boolean
- String
- Xml
- DateTime

2.2.4 API Response & Error Codes
<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OK</td>
</tr>
<tr>
<td>1</td>
<td>Invalid Request</td>
</tr>
<tr>
<td>2</td>
<td>Exception thrown while executing request</td>
</tr>
<tr>
<td>3</td>
<td>Invalid number of parameters supplied</td>
</tr>
<tr>
<td>4</td>
<td>Unsupported parameter type</td>
</tr>
<tr>
<td>5</td>
<td>Invalid Parameter</td>
</tr>
</tbody>
</table>

### 2.2.5 Database Connectivity

The APS framework will host a database connectivity object, described in section 2.8.2, containing all accessor and modifier methods required for querying the database. Additional methods for executing stored procedures will be added as necessary. All database access will be completed using Microsoft’s ADO.NET objects. Furthermore, all database connectivity information will be held in an external xml configuration file to facilitate easy configuration and maintenance.

### 2.2.6 Caching

In effort to reduce load on the APS database, relatively static data such as ride locations will be cached in memory on the framework server. This class, as described in section 2.8.1, will provide access to Microsoft .NET’s internal methods for data caching.

### 2.3 APS Processors

#### 2.3.1 TurnStyleProcessor

The TurnstyleProcessor is a Windows service responsible for notifying the APS framework of people entering and leaving an attraction line. In addition, it is responsible for verifying that a user attempting to enter the QuickTicket line does indeed have a QuickTicket for that time. There will be a TurnstyleProcessor for each attraction.

The processor will notify the APS framework on a set interval of how many people have entered and exited the line in that time period. Communication will be achieved through the RemoteObject.

Furthermore the TurnstyleProcessor will retrieve a list of upcoming QuickTickets for the attraction it is associated with. When a user attempts to enter a QuickTicket line, the processor will first check the locally cached list of QuickTickets to see if the visitor should be granted access. If the QuickTicket is not found in the cached list, the processor will then make a request to the APS framework to validate the visitor’s attempt to access the QuickTicket line.
2.3.2 TextMessageProcessor

The TextMessageProcessor is a Windows service responsible for receiving and parsing text messages that park visitors send to the APS system requesting information. This processor is implemented as a SMTP server to receive text messages from visitors in email format. Visitors are allowed to request information about their group members, pending QuickTickets, and also ride suggestions.

2.3.3 AlertProcessor

The AlertProcessor is the Windows service which is responsible for sending out all alerts. The processor queries the APS framework on a set interval to retrieve all alerts which are set to be sent out for that time period. Upon completion of sending any pending alerts, the processor will sleep for a configurable interval and then repeats the process. Alerts are defined in requirement 3.5.

2.4 Class Relationship Diagram

![Class Relationship Diagram](image)

A - B  B requests action/information from A

2.5 Namespace: APS

2.5.1 Kiosk Class

This class includes the kiosk interface for the APS system as defined in requirement 5.2. Each park kiosk has an instance of Kiosk running on it that retrieves its location and entry flag on instantiation based on its
kioskID. The kiosk caches this information for the day, and uses it when making requests to RemoteObject.

### Kiosk

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_kioskID</td>
<td>int</td>
</tr>
<tr>
<td>m_isEntryKiosk</td>
<td>bool</td>
</tr>
<tr>
<td>m_currentPACID</td>
<td>int</td>
</tr>
<tr>
<td>m_locationID</td>
<td>int</td>
</tr>
<tr>
<td>login(PACCode:string)</td>
<td>bool</td>
</tr>
<tr>
<td>Kiosk(kioskID:int)</td>
<td>void</td>
</tr>
</tbody>
</table>

#### 2.5.1.1 login()

**Description:** Retrieves any schedule, group, and user information available for the supplied PACCode.

**Precondition:** PACCode must be in format as described in Requirements Document. PACCode must be associated with an active user account.

**Postcondition:** User information is returned to the Kiosk. If the PACCode is not authorized to be in the park, admin system is notified.

#### 2.5.2 APSWeb

This class includes the web-based interface for the APS system as defined in requirement 5.1. This is a container class for all Web Interfaces. APSWeb uses RemoteObject to access any Module methods it needs.

### APSWeb

#### 2.5.3 AlertProcessor

This class is responsible for sending out alerts to users as defined in requirement 3.5. AlertProcessor is a timed process that checks on an interval for new Alerts to send to any user, and sends them out as specified by their time in the database.
2.5.4 TextMessageProcessor

This class is responsible for interpreting incoming text message requests to the APS system and returning the requested data. This processor parses user Emails and TextMessages, and sends responses when necessary. Any functionality needed from the Module namespace can be accessed through the RemoteObject.

```
TextMessageProcessor
-processReceivedMessage(): void
-sendResponse(): void
```

2.5.5 TurnstyleProcessor

This class is responsible to maintaining a current count of the number of people in line for a given attraction and updating the APS system via the remote object. Additionally this class is responsible to determining if a person requesting access to the QuickTicket line for an attraction should be granted access.

A Turnstyle Processor exists for each ride in the park and accepts input from every turnstile for that specific ride. The processor is responsible for receiving the input from these individual turnstyles and acting as the middle man between those turnstyles and the rest of the APS Framework.

```
TurnstyleProcessor
-m_attractionID: int
-updateQuickTicketCache(): void
-checkQuickTicket(PACNumber:string): bool
-addPerson(): void
-removePerson(): void
-updateAttractionData(): void
```

2.6 Namespace: APS.Framework

2.6.1 RemoteObject

This class is responsible for interpreting XML requests from the various processors and user interfaces, retrieving the requested data via the APS framework and returning the requested data in XML format.
### 2.6.1.1 `execute()`

<table>
<thead>
<tr>
<th>Description:</th>
<th>Parses the XmlDoc provided and makes the appropriate function calls within the framework, then returns an XmlDoc with the function’s return value.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precondition:</td>
<td>Command:XmlDoc must be in the format described under “API Request Xml” in this document. APS framework must be initialized prior to this method being accessed.</td>
</tr>
<tr>
<td>Postcondition:</td>
<td>The returned XmlDoc response is formatted as described in “API Response Xml”</td>
</tr>
</tbody>
</table>

### 2.7 Namespace: APS.Framework.Modules

#### 2.7.1 User Class

This class holds all methods associated with accessing and modifying the data associated with a user as defined in requirement 4.1 and 4.2. This class includes the methods used by Kiosks and APSWeb to retrieve user data, and change user data in the database.
2.7.1.1  createUser()

**Description:** Inserts a new user into the database

**Precondition:** User does not already exist
PACStartDate <= PACEndDate

**Postcondition:** New entry for the user in the database.
New PAC entered in database if necessary.

2.7.1.2  retrieveUserData()

**Description:** Retrieves all user information from database from a PAC number.

**Precondition:** User must already exist in database.
PAC must be valid.

**Postcondition:** User data is returned.
2.7.1.3  **updateUser()**

**Description:** Updates a user’s information in the database.

**Precondition:** User must already exist in database. PACStartDate <= PACEndDate

**Postcondition:** User entry in database reflects requested changes. New PAC entered in database if necessary.

2.7.2  **Attraction Class**

This class holds all methods associated with accessing and modifying the data associated with an attraction as defined in requirement 4.4. Included in this class are the methods for retrieving and setting attraction info (name, description etc), attraction time configurations, storing attraction history, and the current stored line size.
### 2.7.2.1 updateLineSize()

**Description:** Updates the stored line size for a specific attraction ID

**Precondition:**
- attractionID must be a valid and currently active attraction.
- lineCountChange must be the change in line size since the last update.

**Postcondition:**
Entry for attractionID in database is now += lineCountChange
### 2.7.2.2 visitAttraction()

**Description:**
Creates a new entry in AttractionHistory table, recording the userID, attractionID, and time.

**Precondition:**
- userID must correspond to a user with a valid PAC
- attractionID must correspond to a currently active attraction
- time must not conflict with another entry for the same userID

**Postcondition:**
- New entry exists in AttractionHistory
- Observed preferences of the user have been updated to reflect the visit

### 2.7.2.3 getNumberOfPeopleInLine()

**Description:**
Returns the current line size for a particular attractionID

**Precondition:**
- attractionID must be a currently active attraction

**Postcondition:**
- Line size is unchanged

### 2.7.2.4 getAvailableQuickTicketTimes()

**Description:**
Returns a list of times at which QuickTickets are available for the attractionID

**Precondition:**
- attractionID must be a currently active attraction
- attractionID must have QuickTickets available

**Postcondition:**
- Returns a list of all times that this attractionID still has QuickTickets open for

### 2.7.3 Group Class

This class holds all methods associated with accessing and modifying the data of the group module defined in requirement 3.6. Group includes methods for managing group members and retrieving group statuses.

```
Group

+createGroup(userID:int): int
+addUserToGroup(userID:int, groupId:int, moderator:bool): void
+removeUserFromGroup(userID:int, groupId:int): void
+changeUserModeration(userID:int, moderator:bool): void
+getGroupMembers(groupId:int): Array
```
<table>
<thead>
<tr>
<th>Section</th>
<th>Method</th>
<th>Description</th>
<th>Precondition</th>
<th>Postcondition</th>
</tr>
</thead>
</table>
| 2.7.3.1 | addUserToGroup()        | Adds the userID to the supplied group as a moderator based on the moderator bool.          | userID must not already be assigned to a group  
userID must refer to a currently valid userID with a valid PAC  
groupID must be an existing group                                                                                                                                 | userID is added to groupID as moderator dependant on moderator bool.                                                                                  |
| 2.7.3.2 | removeUserFromGroup()   | Removes the userID from the groupID                                            | groupID must be a valid and active groupID                                                                       | userID is removed from groupID  
If groupID has no more userID with moderator=1 group is disbanded                                                                                     |
| 2.7.3.3 | changeUserModeration()  | Modifies the ability of the userID to moderate the groupID.                    | userID must be assigned to the group already  
userID must be a valid ID with an active PAC  
groupID must be a valid groupID                                                                                                                  |  
If groupID has only one userID with moderate=1 that userID can not be set to moderate=0.  
Otherwise moderate flag for userID is set to reflect the moderate bool                                                                          |
| 2.7.3.4 | getGroupMembers()       | Returns array of group’s aliases and last locations                           | groupID must be a currently active groupID  
Only knows last locations of members that have been to an attractionID                                                                                 | Array of group aliases and locations is returned                                                      |
2.7.4 Schedule Class
This class holds all methods associated with accessing and modifying the data of the suggestion module defined in requirement 3.4. Schedule allows access to the methods that manipulate schedule data within the database and convert schedule entries to QuickTickets.

<table>
<thead>
<tr>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>+addUserScheduleEntry(userID:int, attractionID:int, scheduleTime:datetime, quickTicket:bool): void</td>
</tr>
<tr>
<td>+addGroupScheduleEntry(groupId:int, attractionID:int, scheduleTime:datetime, quickTicket:bool): void</td>
</tr>
<tr>
<td>+removeScheduleEntry(scheduleEntryID:int): void</td>
</tr>
<tr>
<td>+getUserSchedule(userID:int, date:date): Array</td>
</tr>
<tr>
<td>+convertToQuickTicket(scheduleEntryID:int): bool</td>
</tr>
<tr>
<td>-newSchedule(userID:int): bool</td>
</tr>
</tbody>
</table>

2.7.4.1 convertToQuickTicket()

**Description:** Creates a QuickTicket based on the given scheduleEntryID

**Precondition:**
- scheduleEntryID must belong to the requesting userID
- attractionID for the scheduleEntryID must currently be active
- PAC for userID must be valid through the give DateTime
- userID must have available QuickTickets
- userID must be marked as a premiumUser

**Postcondition:** QuickTicket is created for userID

2.7.5 QuickTicket Class
This class holds all methods associated with accessing and modifying the data of the QuickTicket module defined in requirement 3.2. QuickTicket has methods for creating QuickTickets, retrieving QuickTickets, and validating QuickTickets.

<table>
<thead>
<tr>
<th>2.7.5.1 isValid()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> Checks whether a supplied userID has a QuickTicket for the supplied attractionID at the specified time</td>
</tr>
</tbody>
</table>
| **Precondition:**
- userID is a valid and active userID
- attractionID is a valid and currently active attractionID
- QuickTickets are available on attractionID at the specified DateTime
2.7.5.2 getQuickTickets()

**Description:** Retrieves all QuickTickets for userID in the specified range.

**Precondition:** userID must be associated with an active PAC

**Postcondition:** QuickTickets for userID are unchanged and returned in an array

2.7.6 Suggestion Class

This class holds all methods associated with accessing and modifying the data of the suggestion module defined in requirement 3.4. Suggestion allows access to the AI methods that make next ride suggestions, and make schedule filling suggestions.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getNextRides(userID:int, locationID:int, numberRequested:int): int</td>
<td>Suggests the next best ride for userID to take based on line size, preferences, and current location.</td>
</tr>
<tr>
<td>getScheduleSuggestion(userID:int, forDate:date): array</td>
<td>Returns an array of suggested rides in order, based on existing schedule and park history.</td>
</tr>
<tr>
<td>getSuggestion(userID:int, lastLocationID:int, startDate:DateTime, endDate:DateTime): int</td>
<td>Attraction History tables must have sufficient data to produce accurate results.</td>
</tr>
</tbody>
</table>
**Postcondition:** Suggested schedule using attraction history, current schedule, and user preference is returned

### 2.7.7 PAC Class
This class holds all methods associated with accessing and modifying the data of the PAC module defined in requirement 3.1. PAC contains methods that maintain the park’s PACCodes and PACIDs. It also contains the methods for validating PACs.

<table>
<thead>
<tr>
<th>PAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>+createNew(isPremium:bool, startDate:DateTime, endDate:DateTime): int</td>
</tr>
<tr>
<td>+updatePAC(PACID:int, isPremium:bool, startDate:DateTime, endDate:DateTime): bool</td>
</tr>
<tr>
<td>+disablePAC(PACID:int): bool</td>
</tr>
<tr>
<td>+isValid(PACCode:string): bool</td>
</tr>
<tr>
<td>+isValid(PACCode:string): bool</td>
</tr>
</tbody>
</table>

### 2.7.8 Alert Class
This class holds all methods associated with accessing and modifying the data of the alert module defined in requirement 3.5. Class is used by other classes when they need to schedule an alert for a user.

<table>
<thead>
<tr>
<th>Alert</th>
</tr>
</thead>
<tbody>
<tr>
<td>+addAlert(userID:int, text:string, alertTypeID:int, expirationDate:datetime): void</td>
</tr>
<tr>
<td>+removeAlert(alertID:int): void</td>
</tr>
<tr>
<td>+getUserAlerts(userID:int): Array</td>
</tr>
</tbody>
</table>

### 2.8 Namespace: APS.Framework.CommonFunction

#### 2.8.1 CacheFunctions
This class manages any information that needs to be stored in cached memory. Other classes request and set any cached variables they want. This is primarily used for storing important park/attraction state information, to increase access speed and reduce database strain.
2.8.1.1  getCachedObject()

**Description:** Allows access to cached objects

**Precondition:** objName provided must be an existing cached object

**Postcondition:** If objName is cached, than associated object is returned, otherwise null

2.8.1.2  setCachedObject()

**Description:** Adds an object to cache with a key of objName

**Precondition:** objName must not already be associated with a cached object of a different type

**Postcondition:** If objName is already associated with an obj of the same type it is replaced, otherwise false is returned

### DatabaseFunctions

This class contains the methods used for executing simple queries, and retrieving a SqlConnection object when another class needs more precise control over when a connection opens and closes, this helps us cater to the built in .NET connection pooling.

2.8.2.1  getSqlConnection ()

**Description:** Retrieves a SqlConnection object, used to open and close database connections

**Precondition:** Additional connections to the database are available

**Postcondition:** Returns SqlConnection object
2.8.2.2 **executeQuery ()**

*Description:* Executes the supplied SqlCommand, and returns any database rows that result from the query

*Precondition:* The SqlCommand supplied is a valid sql query

*Postcondition:* Changes expected from the SqlCommand have been completed

2.8.2.3 **executeNonQuery ()**

*Description:* This method allows the execution of a query that has no return, generally updates or inserts that don’t return primary keys

*Precondition:* The SqlCommand supplied is a valid nonQuery

*Postcondition:* Changes expected from the SqlCommand have been completed

### 2.9 Sequence Diagrams

2.9.1 **Kiosk Interface Sequence Diagram**

![Kiosk Interface Sequence Diagram](image)

2.9.2 **Kiosk Group Sequence Diagram**
2.9.3 Kiosk Schedule Sequence Diagram

2.9.4 Kiosk Edit User Sequence Diagram
2.9.5 Turnstyle Processor Sequence Diagram

2.9.6 TextMessage Processor Sequence Diagram
3 User Interfaces

3.1 General Overview
This section overviews all the user interfaces the APS system uses. Each section contains a state diagram for all the states of an interface. Though the interface in implementation will have sidebars that lead to all other states, these are not represented on the diagrams in order to reduce clutter and enhance readability.

3.2 Web Interface

3.2.1 Anonymous Web Access
Anonymous Web Access allows users to view maps and ride information. If desired, a ride list is generated through the Ride Information Page. These rides are added to the Ride List Page which is viewable and modifiable. The ride list can then be printed in paper form.
3.2.2 Premium User Web Interface

Premium users retain all functionality of Anonymous Web Interface. Premium Users have login capabilities, accessed through the general web interface. Upon login request, the Premium User is prompted to input his PAC ID and PIN. If access is granted, the Premium User is granted access to Kiosk-Like functionality (Section 3.3). Premium Users can change user settings, view park map, view ride information, create/modify schedules, view list of group members and request suggestions.
3.3 In-Park Kiosk Interface

Users are required to input their PAC via a scanning device. Upon validation, they are allowed access to the Main Screen. The main screen allows access to the following features below.

3.3.1 Access suggestions
Return list of five suggestions based on user

3.3.2 View group members
Show last location of group members, add group members and modify settings.
3.3.3 View Park map

3.3.4 View ride information and wait times
   Add rides to schedule

3.3.5 Configure basic user information
   Name, alerts status and configuration, and groups.

3.3.6 View schedule of rides and QuickTickets.
   Create and modify schedule. Add rides to schedule.

3.4 Mobile Device – Text Interface
The Mobile Device Interface functionality is a limited subset of the In-Park Kiosk functionality. A capable Mobile Device is allowed access to receive alerts, view group member locations, and get request suggestions.
3.5 Mobile Device – PDA Interface

The PDA Interface functionality is a limited subset of the In-Park Kiosk functionality. A configured PDA allows access to receive alerts, view park map, view schedule, schedule rides, use QuickTickets, poll group members, and request suggestions.
3.6 Purchaser Interface

3.6.1 PAC Sales Interface

The PAC Sales user has functionality to activate new accounts, assign PACs, and modify user settings. If a user requires assistance in setting up groups, a PAC Sales user has functionality to create groups.
3.6.2 Purchaser Administrative Interface

The Purchaser Administrative Interface allows the park appointed APS System administrator to modify park and ride specific configuration variables. These settings are broken down into two subsections; a section that requires an administrative login, and a section that does not.
3.6.2.1 General Administrative Interface
The General Administrative Interface corresponds to all functionality that is accessible without an administrative login. These functions include viewing all rides, settings their status, and sending out emergency alerts.

3.6.2.2 Administrative Login Interface
Administrative Login includes all functionality of the General Administrative Interface (section 3.6.2.1) and allows the Purchaser to access additional functionality. The additional functions include adjusting APS system configuration variables like QuickTicket settings, ride times, and park status variables.
4 Data Collection

4.1 General Overview
The databases outlined in the sections below contain all of the data that the APS system needs to operate. The tables encapsulate all aspects of the park: customer data, ride data, and overall/accumulated park data.
4.2 Database Overview Diagram
### 4.3 Table Descriptions

#### 4.3.1 User Table

This table contains the basic data required to keep track of a customer while they’re inside the park.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>userID</td>
<td>A unique integer which tracks a guest while they are in the park</td>
</tr>
<tr>
<td>PACID</td>
<td>A unique integer which tracks the user’s Park Access Card</td>
</tr>
<tr>
<td>groupID</td>
<td>A unique integer shared by all users in the same group</td>
</tr>
<tr>
<td>premiumUser</td>
<td>This bit determines if the user is a premium card holder</td>
</tr>
<tr>
<td>Alias</td>
<td>The name to be displayed on kiosks to identify a user</td>
</tr>
<tr>
<td>mobileEmail</td>
<td>The address to which alerts should be sent, if any</td>
</tr>
<tr>
<td>acceptedAlerts</td>
<td>The alerts, if any, that a user wishes to receive</td>
</tr>
<tr>
<td>groupModerator</td>
<td>Determines if this user is able to make group decisions</td>
</tr>
<tr>
<td>Pin</td>
<td>A card pin number for online access</td>
</tr>
</tbody>
</table>

#### 4.3.2 Attraction History Table

Each table contains the ride history for a particular ride/customer.
4.3.3 Attraction Hour Load History Table
This table contains information to help the park keep track of hourly turnover on an attraction.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attractionLoadHistID</td>
<td>An integer to track an attraction’s person load</td>
</tr>
<tr>
<td>attractionID</td>
<td>A unique integer assigned to a park attraction</td>
</tr>
<tr>
<td>hourOfDay</td>
<td>The hour during the day when the data was collected</td>
</tr>
</tbody>
</table>

4.3.4 Attraction Order History Table
This is a table kept for the management to help them understand the flow of customers through their park.
### Attraction Order History Table

- **attractionOrderHistoryID**: An integer to serve as a lookup key.
- **fromAttractionID**: The attraction ID the person had most recently rode.
- **toAttractionID**: The attraction ID the person visited next.

#### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attractionOrderHistoryID</td>
<td>An integer to serve as a lookup key</td>
</tr>
<tr>
<td>fromAttractionID</td>
<td>The attraction ID the person had most recently rode</td>
</tr>
<tr>
<td>toAttractionID</td>
<td>The attraction ID the person visited next</td>
</tr>
</tbody>
</table>

### 4.3.5 Attraction Table

This table keeps track of ride and attraction information.

- **attractionID**: A unique integer assigned to a park attraction.
- **Name**: The attraction name.
- **description**: A description of the attraction.
- **Intensity**: A numeric rating of the attraction’s intensity.
- **Duration**: The time, in seconds, for which an attraction operates.
- **loadingTime**: The time, in seconds, to move people on and off the ride.
- **Capacity**: The number of people that an attraction can hold.
- **maxQT**: The maximum number of QTs allowable for any one time period.

#### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attractionID</td>
<td>A unique integer assigned to a park attraction</td>
</tr>
<tr>
<td>Name</td>
<td>The attraction name</td>
</tr>
<tr>
<td>description</td>
<td>A description of the attraction</td>
</tr>
<tr>
<td>Intensity</td>
<td>A numeric rating of the attraction’s intensity</td>
</tr>
<tr>
<td>Duration</td>
<td>The time, in seconds, for which an attraction operates</td>
</tr>
<tr>
<td>loadingTime</td>
<td>The time, in seconds, to move people on and off the ride</td>
</tr>
<tr>
<td>Capacity</td>
<td>The number of people that an attraction can hold</td>
</tr>
<tr>
<td>maxQT</td>
<td>The maximum number of QTs allowable for any one time period</td>
</tr>
</tbody>
</table>
maxOverbook | The allowable overbooking of QTs in the case of a group
locationID | A unique identifier for every and any location in the park

4.3.6 Attraction Time
This table contains information concerning the operation times of attractions.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attractionTimeID</td>
<td>The identifier for the current time period</td>
</tr>
<tr>
<td>attractionID</td>
<td>A unique integer assigned to a park attraction</td>
</tr>
<tr>
<td>startTime</td>
<td>The starting/opening time of an attraction</td>
</tr>
<tr>
<td>endTime</td>
<td>The ending/closing time of an attraction</td>
</tr>
<tr>
<td>curDayOfWeek</td>
<td>The day of the week</td>
</tr>
</tbody>
</table>

4.3.7 Attraction Line Table
This table holds up to the minute line information

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attractionLineID</td>
<td>The identifier for the line</td>
</tr>
<tr>
<td>attractionID</td>
<td>A unique integer assigned to a park attraction</td>
</tr>
<tr>
<td>currentWait</td>
<td>A calculated number, in seconds, of the expected line wait</td>
</tr>
<tr>
<td>peopleInLine</td>
<td>The current number of people in an attraction line</td>
</tr>
</tbody>
</table>
4.3.8 Group Table
This holds the list of current active group IDs.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>groupID</td>
<td>A currently active group ID. If no group members remain in the group, it will be removed from the table</td>
</tr>
</tbody>
</table>

4.3.9 PAC Table
This table contains Park Access Card tracking information.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PACID</td>
<td>A unique integer which tracks the user’s Park Access Card</td>
</tr>
<tr>
<td>PACCode</td>
<td>The password for the PAC, if using it externally</td>
</tr>
<tr>
<td>startDate</td>
<td>The date at which the card is counted as active</td>
</tr>
<tr>
<td>endDate</td>
<td>The date at which the card is no longer usable</td>
</tr>
</tbody>
</table>

4.3.10 Schedule Entry Table
This table contains a user’s schedule information.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheduleEntryID</td>
<td>The index into schedules</td>
</tr>
</tbody>
</table>
4.3.11 Alert Table
The alert table is the primary table for alert information. It has its own lookup table, lkAlertType.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alerted</td>
<td>An integer to signify the alert</td>
</tr>
<tr>
<td>userID</td>
<td>A unique integer which tracks a guest while they are in the park</td>
</tr>
<tr>
<td>text</td>
<td>The message to be sent as an alert</td>
</tr>
<tr>
<td>alertTypeID</td>
<td>An identify to categorize alerts types</td>
</tr>
<tr>
<td>expirationTime</td>
<td>The time after which the alert will be irrelevant</td>
</tr>
</tbody>
</table>

4.3.12 lkAlertType Table
This table is an index of possible alert types.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alertTypeID</td>
<td>An identify to categorize alerts types</td>
</tr>
<tr>
<td>description</td>
<td>A description of the alert</td>
</tr>
</tbody>
</table>

4.3.13 Location Table
This table contains the location IDs for every location in the park.
Field Description
---
locationID A unique identifier for every and any location in the park
locationTypeID An identifier for all possible in-park location types

4.3.14 lkLocationType Table
This table holds all possible location types.

Field Description
---
locationTypeID An identifier for all possible in-park location types
description A description for each location type
schedulable This determines if this location can be added to a schedule
attractionStatus This determines if the current attraction is open or closed

4.3.15 Distances Table
This table is an index to finding distances between any two location points in the park.
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>distanceID</td>
<td>The index for the distances</td>
</tr>
<tr>
<td>toLocationID</td>
<td>The initial location</td>
</tr>
<tr>
<td>fromLocationID</td>
<td>The destination location</td>
</tr>
<tr>
<td>distance</td>
<td>The distance between the to and from location IDs</td>
</tr>
</tbody>
</table>

4.3.16 Kiosk Table

This table holds kiosk information. The structure allows kiosks to be moved around the park with minimal work done to keep track of them.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kioskID</td>
<td>The unique identifier for a kiosk</td>
</tr>
<tr>
<td>entryKiosk</td>
<td>This bit determines if this is a designated entry kiosk</td>
</tr>
<tr>
<td>locationID</td>
<td>A unique identifier for every and any location in the park</td>
</tr>
<tr>
<td>IPAddress</td>
<td>The unique IP for the kiosk</td>
</tr>
</tbody>
</table>

4.3.17 Attraction Date Load History Table

Here is the date specific load history.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attractionLoadHistoryID</td>
<td>The identifier for the table</td>
</tr>
<tr>
<td>attractionID</td>
<td>A unique integer assigned to a park attraction</td>
</tr>
<tr>
<td>date</td>
<td>The current date</td>
</tr>
<tr>
<td>count</td>
<td>Total uses of attraction on date</td>
</tr>
</tbody>
</table>
5 Coding Standards

This section details coding standards that must be followed by all developers. It includes standards for naming, style, structure, and .NET specific code.

5.1 Naming Conventions and Style

5.1.1 Camel Casing for method names, local variables, and method arguments

```java
public class myClass
{
    public myMethod(int myArgument)
    {
        string myLocalString;
        int number;
    }
}
```

5.1.2 Prefix private member variables with m_

```java
public class myClass
{
    private int m_number;
    private int m_myMemberString;
}
```

5.1.3 Suffix custom exception classes with Exception

5.1.4 Use verb-object pairs for method names: updateUser()

5.1.5 Use get and a descriptive name for methods with return values: getUserData()

5.1.6 Use strict indentation with Tab (\t)

5.1.7 Comments are indented to the same level as the code they belong to

5.1.8 Descriptive Variable names

5.1.8.1 Avoid Hungarian notation

5.1.8.2 Use long variable names (Do not abbreviate number to num or index to i)
5.1.9 Declare local variables as close to their first use as possible

5.1.10 Declare member variables at the top of class definitions

5.1.11 Declare constants at top of class and use all capitals in constant names

```csharp
public class myClass
{
    private const LEAPYEAR_DAYS = 366;
    public myMethod(int myArgument)
    {
    }
}
```

5.1.12 Only use fully qualified type names when making function calls or declaring a custom variable type, but not for base types

5.1.13 Filenames should reflect the class they contain

5.1.14 Place spaces between operations but not parenthesis (x + y), NOT (x+y)

5.2 Coding Practices

5.2.1 Use basic variable types instead of wrapper classes when possible

5.2.2 Files should only contain information for a single namespace

5.2.3 Wrap lines of code that extend off screen onto the line below starting with an additional `\t`

5.2.4 Document only assumptions and algorithm insights

5.2.5 No magic numbers, hard coded numbers should be constants

5.2.6 Arrays are all zero based

5.2.7 In catch statements that throw exceptions include the original exception to maintain stack traceability

5.2.8 Use resources to store strings that are displayed to users

5.2.9 Use resources to store strings that may change based on deployment environment

5.2.10 No public variables, use modifiers & accessors instead
5.2.11 Place open and close curly braces on new lines
5.2.12 Place single line code blocks after conditionals within curly braces
5.2.13 Parameterized constructors are encouraged
5.2.14 Use base keyword only when invoking a base class constructor or when resolving a specific conflict with a sub class member

5.3 .NET Framework Specific Guidelines

5.3.1 Use Convert class when converting between basic types:
    Convert.ToInt32(myString) instead of myString.ToInt32()

5.3.2 Use regions to encapsulate logical code blocks
    #region Create SqlCommand
    <Code To Create SqlCommand>
    #endregion

5.3.3 Use VS.NET’s automatic documenting comments for function headers
    /// <summary>
    ///
    /// <summary>
    /// <param name="myParam1"></param>
    /// <param name="myParam2"></param>
    /// <returns></returns>

5.3.4 When building long strings use StringBuilder

5.3.5 All code for web pages should be in the code behind aspx.cs file

5.3.6 Always name threads

5.3.7 Avoid using thread priority to set execution control, priority is for semantics

5.3.8 Use Thread.IsAlive() to determine if a thread is still alive
6 Implementation Plan

Setting up the APS System for a given Amusement Park is an involved process; this section serves to set that process into logical steps. These are the recommended steps of implementation, and can be change/rearranged based on the specific needs of a given park.

6.1 Implementation Steps

6.1.1 Compile Park Data

Compiling park data refers to gathering all data relevant to a specific park and processing/encoding it so it is understood by APS.

6.1.1.1 Create APS Database

6.1.1.2 Assign locationIDs to all Kiosks and Attractions

6.1.1.3 Define other areas that may need location IDs and assign IDs to them
This includes meeting areas, food courts, bathrooms, shops etc.

6.1.1.4 For all locationID pairs x and y record the travel distance from x to y
If the distance from y to x is different than the distance from x to y record this information as well. This could happen if a path is only one way. Travel distance refers to the distance required to travel from x to y, no the crow fly’s distance of x to y.

6.1.1.5 Convert existing Attraction history data
This consists of taking any data the park currently has that applies to the APS AttractionHistory tables and entering it into these tables.

6.1.1.6 Fill static and mostly static information into appropriate tables
This includes Attraction load and wait times, and any lifetime visitors the park may have. An example of mostly static information is attraction description.

6.1.2 Configure Park Hardware

Park hardware consists of kiosks, turnstyles and the central machine running the APS framework.

6.1.2.1 Configure Kiosks
Kiosk stores its ID number locally, and makes use of it when making requests from the framework.
6.1.2.2 Configure Turnstyles

Turnstyles store the attraction ID they are for locally and make use of it when making requests to the framework.

6.1.2.3 Other APS configuration files

These files at this point include configuring sql connection strings, timers for processes, and other park/system specific variables.

6.1.3 Install remaining components

After this step all elements of APS should be installed on the appropriate machines.

6.1.3.1 Central Machine: Framework, Database, Processors

6.1.3.2 Exposed Machine: APS web service, TextMessageProcessor

6.1.4 Employee Training

This entails ensuring that the park appointed system administrator and other employees have a basic understanding of their role in APS and also where to find the help manuals and any information they might need. This is decided on a park to park basis entirely, some parks may require extensive training programs while other may just want the manuals.

6.1.5 Profit

6.2 Deployment

After all Implementation Steps have been completed to the Park’s satisfaction, the APS system must be deployed. Deployment involves going live of the new system, while phasing out any existing systems. The initial APS System deployment may require additional park staff to be on hand to help guide customers through using the new interface, as well as to identify and assist in resolving any configuration issues that may arise during the first few days of APS System usage.
# Appendix

## Appendix A - Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADO.NET</strong></td>
<td>Data access component for the Microsoft .NET Framework</td>
</tr>
<tr>
<td><strong>API</strong></td>
<td>Application Interface, in this document API refers to our application</td>
</tr>
<tr>
<td><strong>Request XML</strong></td>
<td>A XML Document with a request sent to RemoteObject, structured as detailed in section 2.3.1</td>
</tr>
<tr>
<td><strong>Response XML</strong></td>
<td>A XML Document with a response sent from RemoteObject back to the Request XML originator, structured as detailed in section 2.3.2</td>
</tr>
<tr>
<td><strong>APS</strong></td>
<td>Abbreviation for Amusement Park Schedule, used in this document it is a reference to the entire APS solution.</td>
</tr>
<tr>
<td><strong>Framework</strong></td>
<td>The namespace given to the encapsulation layer APS uses to facilitate the RemoteObject. Framework can also be a generic reference to the pure processing part of APS, that which doesn't have direct user input.</td>
</tr>
<tr>
<td><strong>Attraction</strong></td>
<td>Defined as any element in the park intended to attract users, includes but not limited to; rides, shows, parades.</td>
</tr>
<tr>
<td><strong>Kiosk</strong></td>
<td>Physical device placed throughout the park that receive input from users and send that to RemoteObject. There is also a Kiosk class that contains many of the functions utilized by all Kiosks.</td>
</tr>
<tr>
<td><strong>PAC</strong></td>
<td>Park Access Card, card required for entry to park and park rides. PAC class contains the functionality required to create and validate PACs.</td>
</tr>
<tr>
<td><strong>PACCode</strong></td>
<td>An alphanumeric string stored in the database in relation to each PACID. This is also the result from reading the barcode/magnetic strip on a PAC, used for easy identification by Kiosks and Turnstyles.</td>
</tr>
<tr>
<td><strong>QuickTicket (QT)</strong></td>
<td>A reserved time slot for a ride. QuickTicket class contains the functionality necessary to receive, use, and validate QuickTickets. QT is an abbreviation used in the database to conserve space.</td>
</tr>
<tr>
<td><strong>RemoteObject</strong></td>
<td>Object class that routes all external requests to the proper method in the proper class.</td>
</tr>
<tr>
<td><strong>Text Message</strong></td>
<td>A string sent to or from a user’s phone. Many cell phones are capable of sending and receiving Text Messages, and convert the messages to an email format for sending. Essentially your phone is just sending and receiving Emails.</td>
</tr>
<tr>
<td><strong>Turnstyle</strong></td>
<td>The APS version of generic turnstiles, the turnstyles have the programming necessary to communicate with the turnstyleProcessor to report line activity.</td>
</tr>
</tbody>
</table>