USER ACCEPTANCE TESTING

By:

AMMAR ZAFAR RAJA

11618519

Table of Contents

BCCCP Software Specification

Entities:

Carpark – implements the ICarpark interface

Constructor - Should take (at least) 3 parameters:

1. A unique name for the carpark (used as an identifier for the carpark.

2. An integer specifying the total capacity of the carpark

3. An integer specifying the number of spaces allocated to season tickets

Throws a Runtime Exception if the name parameter is null or empty Throws a Runtime Exception if the total capacity parameter is less than or equal to zero Throws a Runtime Exception if the number of spaces allocated to season tickets exceeds 10% of the carpark capacity or is less than zero

public void register(ICarparkObserver observer); registers observer as an entity to be notified through the notifyCarparkEvent method when the carpark is full and spaces become available

public void deregister(ICarparkObserver observer); remove observer as an entity to be notified

public String getName(); returns the carpark name

public boolean isFull(); returns a boolean indicating whether the carpark is full (ie no adhoc spaces available)

public IAdhocTicket issueAdhocTicket(); if spaces for adhoc parking are available returns a valid new AdhocTicket throws a RuntimeException if called when carpark is full (ie no adhoc spaces available)

public void recordAdhocTicketEntry(); increments the number of adhoc carpark spaces in use. May cause the carpark to become full (ie all adhoc spaces filled)

public IAdhocTicket getAdhocTicket(String barcode); returns the adhoc ticket identified by the barcode, returns null if the ticket does not exist, or is not current (ie not in use)

public float calculateAddHocTicketCharge(long entryDateTime); returns the charge owing for an adhoc ticket The 'Out-of-Hours' rate of $2/hr should be charged for all the time the car was parked outside of business hours. The 'Business-Hours' rate of $5/hr should be charged for all the time the car was parked during business hours. Business hours are defined as between 7AM and 7PM, Monday to Friday Parking charges are calculated in minute

increments. (IE entry and exit times are truncated to minute intervals) A possible algorithm for calculating charges is given at the end of this document in Appendix 1.

public void recordAdhocTicketExit(); decrements the number of adhoc parking spaces in use. If the carpark is full, For all registered ICarparkObservers call the notifyCarparkEvent method

public void registerSeasonTicket(ISeasonTicket seasonTicket); registers a season ticket with the carpark so that the season ticket may be used to access the carpark throws a RuntimeException if the carpark the season ticket is associated is not the same as the carpark name

public void deregisterSeasonTicket(ISeasonTicket seasonTicket); deregisters the season ticket so that the season ticket may no longer be used to access the carpark

public boolean isSeasonTicketValid(String ticketId); returns true if the season ticket exists, is current (ie the current date is within the period it covers), and the current time is within business hours otherwise returns false

public boolean isSeasonTicketInUse(String ticketId); returns true if the season ticket exists and is currently in use otherwise returns false

public void recordSeasonTicketEntry(String ticketId); causes a new usage record to be created and associated with a season ticket Throws a RuntimeException if the season ticket associated with ticketId does not exist, or is currently in use

public void recordSeasonTicketExit(String ticketId); causes the current usage record of the season ticket associated with ticketID to be finalized. throws throws a RuntimeException if the season ticket associated with ticketId does not exist, or is not currently in use

AdhocTicket – implements IAdhocTicket interface

Constructor - Should take (at least) 3 parameters:

1. A unique number identifying the adhoc ticket

2. A string for the barcode for the ticket. The string should be of the form: “A” + hexstring representation of ticket number + hextring representation of entry date and time.

3. A string for the carpark name.

Throws a RuntimeException if the id number is less than or equal to zero. Throws a RuntimeException if the barcode is empty or null Throws a RuntimeException if the carpark name is empty or null

public int getTicketNo(); returns the unique ticket number of the ticket

public String getBarcode(); returns a unique string identifying the ticket. The string should be of the form: “A” + hexstring representation of ticket number + hextring representation of entry date and time. public String getCarparkId(); returns the string name of the carpark for which the ticket was issued.

public void enter(long dateTime); records the entry date and time provided as milliseconds since the beginning of the epoch. Throws a RuntimeException if dateTime is less than or equal to zero

public long getEntryDateTime(); returns the entry date and time as milliseconds since the beginning of the epoch. returns 0 before an entry is recorded

public boolean isCurrent(); returns true if an entry has been recorded, but an exit has yet to be recorded otherwise returns false

public void pay(long dateTime, float charge); records the date and time of payment provided milliseconds since the beginning of the epoch. records the charge paid for parking throws a RuntimeException if the time of payment is before or equal to the entry time

public long getPaidDateTime(); returns the payment date and time as milliseconds since the beginning of the epoch. returns 0 if the ticket has yet to be paid

public boolean isPaid(); returns true if a payment has been recorded otherwise returns false

public float getCharge(); returns the charge paid for the ticket returns 0 if the charge has yet to be paid public void exit(long dateTime); records the final exit time for the ticket provided as milliseconds since the beginning of the epoch. throws a RuntimeException if the exit time is before or equal to the payment time

public long getExitDateTime(); returns the exit date and time as milliseconds since the beginning of the epoch. returns 0 if the ticket has yet to exit

public boolean hasExited(); returns true if the ticket has exited otherwise returns false

Hint: Possible AdhocTicket states: ISSUED, CURRENT, PAID, EXITED

SeasonTicket – implements the ISeasonTicket interface

Constructor - Should take (at least) 4 parameters:

1. A unique string identifying the season ticket. The string should be of the form: "S" + hexstring representation of a unique season ticket number

2. A string identifying and exactly equivalent to the carpark name of the carpark for which the season ticket is issued.

3. A long specifying the starting date and time of the period for which the season ticket is issued as milliseconds since the beginning of the epoch. 4. A long specifying the end date and time of the period for which the season ticket is issued as milliseconds since the beginning of the epoch

Throws a RuntimeException if the ticket id string is empty or null. Throws a RuntimeException if the carpark name is empty or null Throws a RuntimeException if the starting date is less than or equal to zero Throws a RuntimeException if the end date is less than or equal to the starting date

public String getId(); returns the unique string identifying the SeasonTicket.

public String getCarparkId(); returns the unique name idenfifying the carpark associated with the season ticket

public long getStartValidPeriod(); returns a date and time for the beginning of when the season ticket is valid as milliseconds since the beginning of the epoch

public long getEndValidPeriod(); returns a date and time for the beginning of when the season ticket is valid as milliseconds since the beginning of the epoch

public boolean inUse(); returns true if a UsageRecord is current otherwise returns false

public void recordUsage(IUsageRecord record); records a new UsageRecord as current throws a RuntimeException if UsageRecord is null

public IUsageRecord getCurrentUsageRecord(); returns the current usage record if the season ticket is in use otherwise returns null

public void endUsage(long dateTime); records a time for the end of the current usage in the current UsageRecord throws a RuntimeError if the season ticket is not currently in use throws a RuntimeError if the specified end dateTime is less than or equal to the starting time of the current UsageRecord

public List<IUsageRecord> getUsageRecords(); returns a List of all UsageRecords recorded for the season ticket returns an empty list if no usages have been recorded

Appendix 1

Algorithm for calculating the parking charge

Start and end are assumed to be in milliseconds since the beginning of the epoch startBH and endBH represent the beginning and end of business hours on a business day BH_Rate represents the Business Hour rate (be careful with units) OOH_Rate represents the Out of Hour rate

Some details such as how to calculate the time value for 'midnight', how to calculate 'nextDay', and how to check whether a day is a business day are omitted.

calcCharge(start, end)

startTime = start.Time truncated to previous minute endTime = end.Time trunated to previous minute

curDay = startTime.Day endDay = endTime.Day

charge = 0 curStartTime = startTime while curDay != endDay curEndTime = curDay.midnight charge += calcDayCharge(curStartTime, curEndTime, curDay) curStartTime = curEndTime curDay = curDay.nextDay

charge += calcDayCharge(curStartTime, endTime, endDay) return charge

calcDayCharge(startTime, endTime, day)

dayCharge = 0 if isBusinessDay(day) if endTime <= startBH dayCharge = (endTime - startTime) * OOH_Rate

else if startTime >= endBH dayCharge = (endTime - startTime) * OOH_Rate

else if startTime >= startBH and endTime <= endBH dayCharge = (endTime - startTime) * BH_Rate

else if startTime < startBH and endTime <= endBH dayCharge = (startBH - startTime) * OOH_Rate dayCharge += (endTime - startBH) * BH_Rate

else if startTime >= startBH and startTime < endBH and endTime > endBH dayCharge = (endBH - startTime) * BH_Rate dayCharge += (endTime - endBH) * OOH_Rate

else if startTime < startBH and endTime > endBH dayCharge = (startBH - startTime) * OOH_Rate dayCharge += (endBH - startBH) * BH_Rate dayCharge += (endTime - endBH) * OOH_Rate else error else dayCharge = (endTime - startTime) * OOH_Rate

return dayCharge