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 RuntimeException if the name parameter is null or empty
Throws a RuntimeException if the total capacity parameter is less than or equal to zero
Throws a RuntimeException 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:

4. A unique number identifying the adhoc ticket
5. 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.
6. 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

UsageRecord – implements the IUsageRecord interface

Constructor – should take 2 parameters

1. A string identifying the associated season ticket
2. A long specifying the start date and time in milliseconds since the beginning of the epoch

Throws a RuntimeException if the string identifying the season ticket is null or empty
Throws a RuntimeException if the start time is less than or equal to zero

public void finalise(**long** endDateTime);
records the end time for the carpark usage
throws a RuntimeException if the end time is less than or equal to zero

public long getStartTime();
returns the start time for the usage

public long getEndTime();
returns the end time for the usage
returns 0 if the end time has yet to be set

public String getSeasonTicketId();
returns the associated season ticket id

DAOs and Factories

AdhocTicketDAO – implements IAdhocTicketDAO

Constructor – should take 1 parameters

1. An instance of a class implementing the IAdhocTicketFactory

Throws a RuntimeException if the reference to the adhocTicketFactory is null

public IAdhocTicket createTicket(String carparkId);
Returns a valid adhoc ticket
Throws a RuntimeException if carparkID is null or empty

public IAdhocTicket findTicketByBarcode(String barcode);
returns the adhoc ticket identified by barcode
returns null if the ticket does not exist

public List<IAdhocTicket> getCurrentTickets();
returns a List of current adhoc tickets (tickets representing cars currently parked)
returns an empty list if no adhoc tickets are currently in use
`

AdhocTicketFactory – implements IAdhocTicketFactory

public IAdhocTicket make(String carparkId, **int** ticketNo);
returns a valid instance of a class supporting the IAdhocTicket interface

throws a RuntimeException if carparkId is null or empty
throws a RuntimeException if ticketNo is less than or equal to zero

SeasonTicketDAO – implements ISeasonTicketDAO

Constructor – should take 1 parameters

1. An instance of a class implementing the IUsageRecordFactory

throws a RuntimeException if the reference to the UsageRecordFactory is null

public void registerTicket(ISeasonTicket ticket);
records an instance of a season ticket in an internal store
throws a RuntimeException if ticket is null

public void deregisterTicket(ISeasonTicket ticket);
removes an instance of a season ticket from the internal store
throws a RuntimeException if ticket is null

public int getNumberOfTickets();
returns the number of season tickets currently registered in the internal store

public ISeasonTicket findTicketById(String ticketId);
returns a season ticket identified by ticketId
returns null if the season ticket is not found

public void recordTicketEntry(String ticketId);
adds a valid UsageRecord to the season ticket identified by ticketId
throws a RuntimeException if the season ticket identified by ticketId is not in the internal store

public void recordTicketExit(String ticketId);
finalises the current usage of the season ticket identified by ticketId
throws a RuntimeException if the season ticket identified by ticketId is not in the internal store
throws a RuntimeException if the season ticket identified by ticketId is not currently in use

UsageRecordFactory – implements IUsageRecordFactory

public IUsageRecord make(String ticketId, **long** startDateTime);
returns a valid instance of a class supporting the IUsageRecord interface
throws a RuntimeException if ticketId is null or empty
throws a RuntimeException is less than or equal to zero

Controllers

EntryController – implements the IEntryController, ICarSensorResponder, and ICarparkObserver interfaces

NOTE: the following assumes that the following states illustrated in the state diagram have been implemented IDLE, WAITING, FULL, BLOCKED, TICKET_ISSUED, TICKET_VALIDATED, TICKET_TAKEN, ENTERING, ENTERED

IDLE – no car detected on any sensor, waiting for a car to arrive.
WAITING – a car is detected on the outside sensor, waiting for the button to be pushed or a season ticket to be inserted
FULL – a car is still detected on the outside sensor, the button has been pushed, but no adhoc spaces are available
BLOCKED – a car is detected on the inside sensor, while another car is still detected on the outside sensor but the barrier is down (ie a car is in the process of entering, but has yet to take or retrieve their ticket
TICKET_ISSUED - a car is still detected on the outside sensor, and a ticket has been issued after the button was pushed. Waiting for the ticket to be taken
TICKET_VALIDATED – a car is still detected on the outside sensor, and a season ticket has been ejected after being validated. Waiting for the season ticket to be retrieved
TICKET_TAKEN – a car is still detected on the outside sensor, the adhoc or season ticket has been taken, and the entry barrier is raised. Waiting for the car to progress through the entry barrier
ENTERING – a car is detected on the inside sensor, while still being detected on the outside sensor (ie the car is under the barrier). The barrier is still up. Waiting for the car to clear the barrier.
ENTERED – the car is still detected on the inside sensor, but has cleared the outside sensor. The barrier is still up. Waiting for the car to clear the entry

Constructor – should take 5 parameters

1. An instance of a class implementing the ICarpark interface representing the carpark
2. An instance of a class implementing the IGate interface representing the entry gate
3. An instance of a class implementing the ICarSensor interfaces representing the outside car sensor
4. An instance of a class implementing the ICarSensor interfaces representing the inside car sensor
5. An instance of a class implementing the IEntryUI interface representing the control pillar user interface

throws a RuntimeException if the reference to any of these objects is null

The constructor should:
1. Register the entry controller with the carpark as an ICarparkObserver
2. Register the entry controller with both car sensors as a ICarEventResponder
3. Register the entry controller with the ui as and IEntryController
4. Initialise the entry controller state to IDLE

public void notifyCarparkEvent();
called when the carpark detects a ‘carpark event’ – ie when the carpark is full and a car leaves
checks with carpark whether any adhoc spaces are available (ie whether carpark ‘full’)
if spaces available
    display ‘Push Button’ message
    transitions to WAITING state
otherwise
    do nothing

public void buttonPushed();
response depends on controller state
if the controller is in the WAITING state:
    check whether carpark is full (no adhoc spaces available)
    if carpark full
        displays ‘Carpark Full’ message
        transitions to ‘FULL’ state
    otherwise
        issue a new adhoc ticket
        display ‘Take Ticket’ message
        transition to ‘TICKET_ISSUED’ state
otherwise
    cause UI to ‘beep’

public void ticketInserted(String barcode);

response depends on controller state
if controller in WAITING state:
    check whether the season ticket is valid
    check whether the season ticket is in use
    if season ticket is valid and not currently in use
        record the barcode for later entry processing
        display ‘Take Ticket’ message
        transition to the TICKET_VALIDATED state
    otherwise
        cause the UI to ‘beep’
otherwise
    cause the UI to ‘beep’

public void ticketTaken();
response depends on controller state
if controller in TICKET_ISSUED or TICKET_VALIDATED states
    clear ‘Take Ticket’ message (optional: display ‘Ticket Taken’ message)
    raise the entry gate
    transition to ‘TICKET_TAKEN’ state
otherwise
    cause the UI to ‘beep’

public void carEventDetected(String detectorId, **boolean** detected);

called when a car sensor detects a car event – either a car is newly detected, or a car absence is newly detected
The response depends on the EntryController state and which sensor detects the event..

IDLE – outside sensor detects car presence
        Display ‘Push Button’ message
        Controller transitions to WAITING state
    – inside sensor detects car presence
        Controller transitions to BLOCKED state

WAITING, FULL, TICKET_ISSUED, TICKET_VALIDATED
    – outside sensor detects car absence
        Controller transitions to IDLE state
    – inside sensor detects car presence
        Controller transitions to BLOCKED state

BLOCKED – outside sensor detects car absence
        Controller transitions to IDLE state
    – inside sensor detects car absence
        Controller transitions to previous state (display appropriate message)

TICKET_TAKEN – outside sensor detects car absence
        Controller transitions to IDLE state
    – inside sensor detects car presence
        Controller transitions to ENTERING state

ENTERING – outside sensor detects car absence
        Controller transitions to ENTERED state
    – inside sensor detects car absence
        Controller transitions back to TICKET_TAKEN state

ENTERED – inside sensor detects car absence
        Lowers the entry gate
        Controller transitions to IDLE state
    – outside sensor detects car presence
        Controller transitions back to ENTERING state

ExitController – implements the IExitController, ICarSensorResponder interfaces

NOTE: the following assumes that the following states illustrated in the state diagram have been implemented:
IDLE, WAITING, BLOCKED, PROCESSED, REJECTED. TAKEN, EXITING, EXITED

IDLE – no car detected on any sensor, waiting for a car to leave.
WAITING – a car detected on the inside sensor, waiting for an adhoc or season ticket to be inserted
BLOCKED – a car is detected on the outside sensor, while another car is in the process of exiting but the barrier is still down
PROCESSED - a car is still detected on the inside sensor, and the ticket has been ejected after being confirmed as paid, or confirmed as a valid season ticket currently in use. Waiting for the ticket to be taken.
REJECTED - a car is still detected on the inside sensor, and the ticket has been ejected after being rejected as not paid, unreadable, or rejected as not a valid season ticket currently in use. Waiting for the ticket to be taken.
TAKEN – a car is still detected on the inside sensor, the adhoc or season ticket has been taken, and the exit barrier is raised. Waiting for the car to progress through the exit barrier.
EXITING – the car is detected on the outside sensor, while still being detected on the inside sensor (ie the car is under the barrier). The barrier is still up. Waiting for the car to clear the barrier.
EXITED – the car is still detected on the outside sensor, but has cleared the inside sensor. The barrier is still up. Waiting for the car to clear the exit.

Constructor – should take 5 parameters

1. An instance of a class implementing the ICarpark interface representing the carpark
2. An instance of a class implementing the IGate interface representing the entry gate
3. An instance of a class implementing the ICarSensor interfaces representing the inside car sensor
4. An instance of a class implementing the ICarSensor interfaces representing the outside car sensor
5. An instance of a class implementing the IExitUI interface representing the control pillar user interface

throws a RuntimeException if the reference to any of these objects is null

The constructor should:
1. Register the exit controller with both car sensors as a ICarEventResponder
2. Register the exit controller with the ui as and IExitController
3. Initialise the exit controller state to IDLE

public void carEventDetected(String detectorId, **boolean** detected);
called when a car sensor detects a car event – either a car is newly detected, or a car absence is newly detected
The response depends on the ExitController state, and which sensor detects the event.

IDLE – inside sensor detects car presence
Display ‘Insert Ticket’ message
Controller transitions to WAITING state
– outside sensor detects car presence
Controller transitions to BLOCKED state

WAITING, FULL, TICKET_ISSUED, TICKET_VALIDATED
– inside sensor detects car absence
Controller transitions to IDLE state
– outside sensor detects car presence
Controller transitions to BLOCKED state

BLOCKED – inside sensor detects car absence
Controller transitions to IDLE state
– outside sensor detects car absence
Controller transitions to previous state (display appropriate message)

TAKEN – inside sensor detects car absence
Controller transitions to IDLE state
– outside sensor detects car presence
Controller transitions to EXITING state

EXITING – inside sensor detects car absence
Controller transitions to EXITED state
– outside sensor detects car absence
Controller transitions back to TAKEN state

EXITED – outside sensor detects car absence
Controller transitions to IDLE state
– inside sensor detects car presence
Controller transitions back to EXITING state

```
public void ticketInserted(String ticketStr);
response depends on controller state, and the value of ticketStr
if controller is in the WAITING state
        check whether the barcode is a valid barcode
        if the barcode starts with 'A'
                find the adhoc ticket by barcode
                if the adhoc exists and the ticket is current and has been paid
                        record the exit time for the adhoc ticket
                        display 'take processed ticket'
                        transition controller to 'PROCESSED' state
                otherwise
                        display 'take rejected ticket'
                        transition controller to REJECTED state
        else if barcode starts with 'S'
                find the season ticket by barcode (ticketID)
                if the season ticket is registered, and is valid, and is currently in use
                        finalise the season tickets current usage record
                        display 'take processed ticket'
                        transition controller to 'PROCESSED' state
                otherwise
                        display 'take rejected ticket'
                        transition controller to REJECTED state
        otherwise
                display 'take rejected ticket'
                transition controller to REJECTED state
                make the UI 'beep'
otherwise
        make the UI 'beep'


public void ticketTaken();
response depends on controller state
if controller is in the PROCESSED state
        raise the exit gate
        transition controller to TAKEN state
else if controller is in the REJECTED state
        transition controller to WAITING state
otherwise
        make the UI 'beep'
```

PaystationController – implements the IPaystationController interface

NOTE: the following assumes that the following states illustrated in the state diagram have been implemented
IDLE, WAITING, REJECTED, PAID

IDLE – no processing current, waiting for a ticket to be inserted
WAITING – a valid ticket has been inserted and a charge has been calculated and displayed,
waiting for the ticket to be paid
REJECTED – an invalid ticket was inserted, waiting for the ticket to be taken
PAID – a valid ticket has been paid, waiting for the ticket to be taken

Constructor – should take 2 parameters

1. An instance of a class implementing the ICarpark interface representing the carpark
2. An instance of a class implementing the IPaystationUI interface representing the user interface for the paystation

Throws a RuntimeException if either of the parameters is null

The constructor should:
1. Register the controller with the UI as an IPaystationController
2. Initialise the paystation controller state to IDLE

```
public void ticketInserted(String barcode);
if the controller state is IDLE
        request the carpark to return the adhoc ticket identified by the barcode
        if a ticket is returned and is current and not paid
                request the carpark to calculate the charge
                display the charge
                transition controller to WAITING state
        otherwise
                make the UI 'beep'
                transition controller to REJECTED state
otherwise
        make the UI 'beep'
```

```
public void ticketPaid();
if the controller is in the WAITING state
        record the payment time and charge for the adhoc ticket
        print payment details on ticket
        transition controller to PAID state
otherwise
        make the UI 'beep'
```

```
public void ticketTaken();
if the controller is in the WAITING, PAID, or REJECTED states
        transition the controller to the IDLE state
otherwise
        make the UI 'beep'
```

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
```