University Car Park

# Design Overview

Our design contains a main CarPark class which other classes and methods interact with. Each entrance and exit to the carpark is implemented as a separate thread which adds and removes cars from the car park accordingly. Separate threads then control the proportion of cars entering vs cars leaving as well as managing cars that are in the carpark but have not found a space yet.

# Class Design

### Car

Encapsulates car attributes. Currently only used to check if the driver is considerate or not, ie: are they going to park across two spaces?

### WaitManager

Acts as a wrapper for an ArrayList that holds cars currently inside the car park but still searching for a space. Allows normal ArrayList operations as well as passing the list by reference to different threads. This class holds any cars that are in the carpark when there are no spaces available.

### Parker

Constantly tries to empty the WaitManager queue into the car park. Calls the park method on the CarPark class which blocks until there is a space available.

### Clock

Maintains a counter that is used when checking how long entrance and exit threads should sleep for (and therefore control the ratio of how many cars are entering vs how many are leaving).

### Entrance

Adds cars to the WaitManager queue. If the total number of cars in the carpark exceeds the maximum capacity of the carpark then the entrance generates a float between 0 and 1. It then compares this number to the entry chance, which is 1 divided by the number of cars in excess of capacity. If the number is less than this ratio, then the entrance will still queue the car. This ratio halves with every successive car that is added to the carpark, and so the chance of adding more cars when over capacity should fall dramatically.

### Exit

Removes cars from the carpark. Also occasionally introduces delays to mimic the effect of someone forgetting their ticket, etc…

### CarPark

Tracks number of spaces, spaces occupied and the cars occupying them. Provides a number of getter methods for getting information on the car park and its contents.

# Concurrency Considerations

Entrance threads require a lock on the WaitManager in order to fill it but otherwise do not interact with the other threads. Their only consideration on whether to fill the WaitManager or not is whether there is enough potential room in the car park to accommodate more cars. They require a lock on the CarPark object to determine this, but immediately release it once they have the answer.

Parker threads require a lock on the CarPark in order to insert cars into the CarPark and so it cannot insert cars at the same time as an Exit thread is trying to remove one and vice versa. The park method may block waiting for a space. If it does it releases the lock on the CarPark object with a wait() call, allowing the Exit threads to free up some spaces before it tries again.

Exit threads require a lock on the CarPark object in order to remove cars from the CarPark. They will block if there are no cars to remove but should otherwise have equal access to cpu time compared to the other threads.

The Clock thread manages the timer which modifies the amount that the Entrance and Exit threads sleep for after each iteration through the loop. This governs when there are more cars entering the car park vs exiting and vice versa. Since the Clock has getter methods only and the values cannot be modified from outside the thread it is safe for multiple threads to access the Clock’s values simultaneously.