Assignment 3

Formalities

This assignment may be solved in groups of up to 3 members. Each member must hand in themselves. Attach a file naming the group members.

Deadline is Friday the 30th of November.

This assignment must be handed in and accepted, for you to go to the exam. At the exam we will talk about parts of your assignments.

Topics for this assignment:

- Readers & Writers
- Proxy
- Flyweight
- Unit testing
- Multi-threading

Introduction

In this assignment you're going to simulate citizens of a city, BridgeVille, going to work outside the city. To get outside, they need to cross a single lane bridge. When exiting the city, they go through a check point, and just give their ID. When entering again they must give more information about the car.

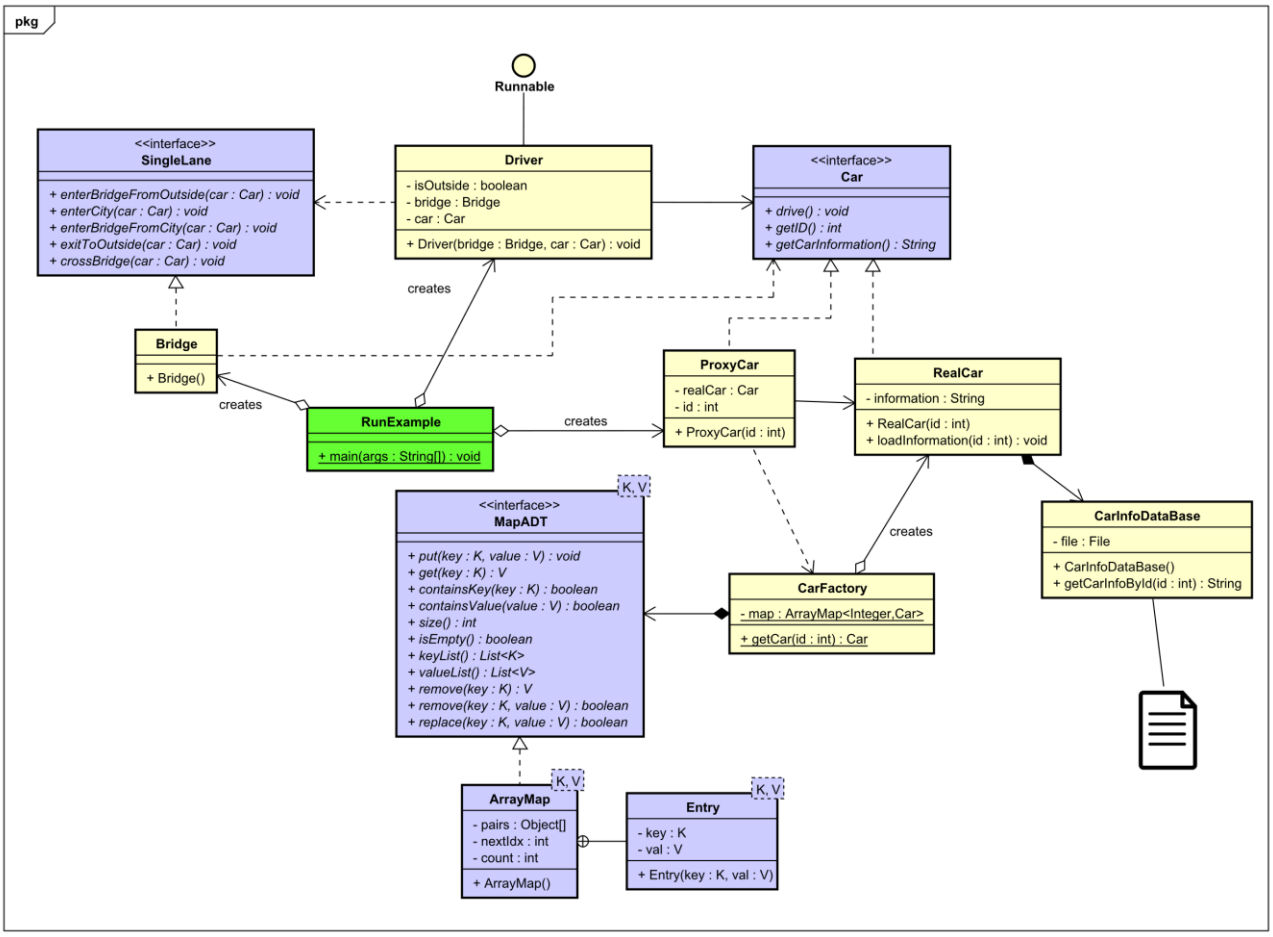
We must figure out how to let cars cross the bridge, and how to retrieve the extra information about the drivers.

There are only 10 types of cars allowed in BridgeVille, and we must keep track of information about the Car type a specific Driver drives.

UML

Below is a UML diagram. Again, you're allowed to add methods and variables to the classes, or change names.

You'll be given some classes, as well as a file with some information about cars.



The purple classes are given to you.

The yellow classes, you must implement.

The green class contains the main method to run the example

Unit testing

You must perform unit testing of the ArrayMap. At the end of the document is documentation for how the ArrayMap works.

I suggest you look through the powerpoint from the unit testing session again. Consider for example how to perform

- Path testing (test all branches of if-else statements)
- Equivalence partitioning
- Boundary analysis
- Etc..

You're not required to thoroughly test all methods, but just enough that you can show and explain different examples and strategies for unit testing.

Implementation

Below is a description of each of the classes, you must implement.

Bridge

This class is a slightly different version of the Readers/Writers problem. Instead of having readers and writers, you have "cars from the city" and "cars from the outside". But the concept is just the same. Furthermore, instead of allowing only one Writer/"car from the outside" access at a time, you can allow multiple. So it will be like having two versions of readers.

You decide which strategy you want to implement, e.g. giving preference to one side over the other, or using a balanced approach, or a fair.

The Bridge is considered to be the shared resource. The method "CrossBridge" represents a car crossing the bridge (or a Reader reading the resource). Include here a Thread.sleep(…), to represent that it takes some time to cross the bridge.

So, when a Driver wants to cross the Bridge from e.g. the city side to outside, it will call

- enterBridgeFromCity(car) ← It requests access to the shared resource. It may have to wait()
- crossBridge(car) ← The driver crosses the bridge
- exitBridgeToOutside(car) ← The driver notifies that it is done using the bridge.

And vice versa, if a Driver wishes to enter the city, it will call

- enterBridgeFromOutside(car) ← Request access to the Bridge, it may have to wait(), if the Bridge is in use from the other side
- crossBridge(car) ← cross the bridge
- enterCity(car) ← notify that the Driver no longer uses the Bridge.

Include appropriate print outs, so you can see what's going on. This is what the 'car' parameter is for, you can give each Driver a name or an id, and you can use the Car's ID as well.

NOTE: in the method Bridge::enterCity(car : Car), you must call the method car.getCarInformtation(). This represents that the Driver shows the extra information to the border guard. This will kick off the lazy instantiation of the ProxyPattern.

Driver

This class implements Runnable, so it's a thread. The constructor takes an instance of Bridge, so the Driver can access the Bridge. This instance of Bridge is shared between all Drivers.

In the run() method, have a while-true-loop. The Driver is supposed to represent a citizen from BridgeVille going to work outside. So first the Driver crosses the Bridge from city to outside. Then it turns around and cross the Bridge from outside to the city. All the Drivers keep going back and forth. You could use a boolean to figure out which side the Driver should enter the Bridge from, see the assignment presentation slides.

The Driver has an instance of Car. I.e. the Driver doesn't care whether it's a ProxyCar or a RealCar.

ProxyCar

This class implements the Car interface, and is part of the Proxy pattern. We use this to do lazy instantiation of the RealCar. The ProxyCar has an id in the range [0 , 9].

When the getCarInformation() method is called, this ProxyCar class must get a RealCar instance from the CarFactory, based on the ID.

RealCar

This class implements the Car interface, and is part of the Proxy pattern. When this class is instantiated, you use the CarInfoDatabase to load information about the Car, based on the id of the RealCar. This represents that the RealCar class is heavy to instantiate, and so we only want to load the information, when it's actually needed.

CarFactory

This class is part of the Flyweight design pattern. It must instantiate RealCar objects, based on the id provided. It uses to provided ArrayMap to hold the created objects. Notice the getCar(id : int) method is static

CarInfoDatabase

This class reads lines from the provided txt file. You can see in the assignment presentation how to implement the getCarInfoByID(id : int)-method.

RunExample

This class contains the main method. First you instantiate a Bridge instance. Then you create a bunch of Drivers, and for each Driver you provide it with a ProxyCar. The Drivers should then drive back and forth across the Bridge. See if you can let some Drivers start outside of the city, and some Drivers start inside the city.

Map Documentation

Note: The map does not allow null values, neither as Key nor as Value.

Interface MapADT<K,V>

- **Type Parameters:**

  `K - the type of keys maintained by this map`

  `V - the type of mapped values`

put

`void put(K key, V value)`

Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced by the specified value. (A map `m` is said to contain a mapping for a key `k` if and only if `m.containsKey(k)` would return `true`.)

Parameters:

`key - key with which the specified value is to be associated`

`value - value to be associated with the specified key`

Throws:

`NullPointerException - if the specified key or value is null and this map does not permit null keys or values`

get

`V get(Object key)`

Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.

More formally, if this map contains a mapping from a key `k` to a value `v` such that `key.equals(k)`, then this method returns `v`; otherwise it returns `null`. (There can be at most one such mapping.)

Parameters:

`key - the key whose associated value is to be returned`

Returns:

`the value to which the specified key is mapped, or null if this map contains no mapping for the key`

Throws:

`NullPointerException - if the specified key is null and this map does not permit null keys (optional)`

ContainsKey

`boolean containsKey(Object key)`

Returns `true` if this map contains a mapping for the specified key. More formally, returns `true` if and only if this map contains a mapping for a key `k` such that `key.equals(k)`. (There can be at most one such mapping.)

Parameters:

key - key whose presence in this map is to be tested

Returns:

true if this map contains a mapping for the specified key

Throws:

NullPointerException - if the specified key is null and this map does not permit null keys (optional)

containsValue

`boolean containsValue(Object value)`

Returns `true` if this map maps one or more keys to the specified value. More formally, returns `true` if and only if this map contains at least one mapping to a value `v` such that `value.equals(v)`.

Parameters:

value - value whose presence in this map is to be tested

Returns:

true if this map maps one or more keys to the specified value

Throws:

NullPointerException - if the specified value is null and this map does not permit null values (optional)

size

`int size()`

Returns the number of key-value mappings in this map.

Returns:

the number of key-value mappings in this map

isEmpty

`boolean isEmpty()`

Returns `true` if this map contains no key-value mappings.

Returns:

true if this map contains no key-value mappings

keyList

`List<K> keyList()`

Returns a `List` view of the keys contained in this map.

Returns:

a list view of the keys contained in this map

valueList

`List<V> valueList()`

Returns a `List` view of the values contained in this map.

Returns:

a list view of the values contained in this map

remove

`V remove(Object key)`

Removes the mapping for a key from this map if it is present. More formally, if this map contains a mapping from key `k` to value `v` such that `key.equals(k)`, that mapping is removed. (The map can contain at most one such mapping.)

Returns the value to which this map previously associated the key, or `null` if the map contained no mapping for the key.

The map will not contain a mapping for the specified key once the call returns.

Parameters:

key - key whose mapping is to be removed from the map

Returns:

the previous value associated with key, or null if there was no mapping for key.

Throws:

NullPointerException - if the specified key is null and this map does not permit null keys (optional)

remove

`boolean remove(Object key, Object value)`

Removes the entry for the specified key only if it is currently mapped to the specified value.

Parameters:

key - key with which the specified value is associated

value - value expected to be associated with the specified key

Returns:

true if the value was removed

Throws:

NullPointerException - if the specified key or value is null, and this map does not permit null keys or values (optional)

replace

`boolean replace(K key, V value)`

Replaces the entry for the specified key only if it is currently mapped to some value.

Parameters:

key - key with which the specified value is associated

value - value to be associated with the specified key

Returns:

True if the value of key was replaced. Otherwise false

Throws:

NullPointerException - if the specified key or value is null, and this map does not permit null keys or values