# Report and Investigation

## Coupling and Cohesion

Coupling is a measure of connectedness between two modules, where a lower coupling is deemed favourable. Classes must, however, pertain at least some degree of coupling to communicate. Cohesion is the measure of independence which a class, where higher cohesion is considered best. A class with high cohesion will perform a single action, or achieve a single goal, as to reduce the amount of unnecessary data. Hence, these two things will be at odds, with a high cohesion resulting in a low coupling.

Coupling is measured using three key points; size, intimacy, and flexibility. Size refers to the number of connections between routines. Size of a coupling should be kept to a minimum, a loose coupling, else an increased information flow between objects may slow down the program. Tightly coupled classes will suffer from the “ripple effect”, in which a change in one class will require changes in other classes making them difficult to edit and maintain and may result in difficulty understanding the class without referencing it’s connected classes. Intimacy of a coupling is the directedness of the connections between classes. Flexibility is a measure of how easy it is to change a connection between two classes. Inheritance coupling focuses on hierarchal inheritance of classes to certify efficiency by maximising variable inheritance, so that children will not have duplicate variables leading to an inefficient and poorly organised program.

Cohesive classes tend to do a single job particularly well, and all elements of the class have a relation to each other. Methods within classes should also be cohesive to maximise efficiency, which is to say unnecessary code shouldn’t be added that performs other tasks, rather this code should be given its own method. Having a high cohesion allows for greater understanding of a class’ purpose, and the purpose of it’s contained methods, and allows meaningful names to be given to classes. Reusability of classes is also a key advantage of cohesive class design, as it allows for code to be reused rather than duplicated.

There are several types of cohesion, including operation cohesion, class cohesion and specialisation cohesion. Operational cohesion ensures that an operation will focus on a single requirement, such as a single calculation or return. Class cohesion, like operational, will measure how a class focuses on a single requirement, as to omit unnecessary variables that could be contained in their own respective classes. Specialisation cohesion considers the appropriateness of inheritance, in instances where simply linking classes with associations is more relevant.

The advantages of these design features were considered while creating our design work to produce a clear and efficient implementation of the provided scenario. Interaction coupling was considered in method creation to pass objects rather than several variables. Inheritance coupling, and specialisation cohesion were considered when linking classes, for example Journal will inherit the features of an item, while adding its own distinct methods and variables, however a book class was not created as it’s inheritance would be inefficient due to having no additional featured compared to an item.

## Implementations of List

Lists are one of the more prominent Java collections. Two of the three implementations of Lists in the collections framework are ArrayLists and LinkedLists, which each offer their own performance advantages over one another.

In an ArrayList, elements are stored contiguously starting from index 0. They offer an advantage over standard arrays, since an array cannot be resized once created, where a List implementation allows indefinite expansion. ArrayLists will use a backing array that contains all array elements, along with a method of tracking List size. Once an ArrayList’s capacity is reached, a new, larger array will replace the backing array, typically twice the capacity of the previous, containing all data from the previous array.

Getting an element in an ArrayList is made quick due to the contiguous storage, so getting the *n*th element can be done by simply skipping straight to index *n*, meaning it is effective for random access of elements. To insert an element at the end of an ArrayList it can simply be added to the end of the array, at postion *n+1* where *n* is the size of the ArrayList. Complexity comes from inserting elements into the middle of the array, as all following elements must be moved 1 index along to create space for the new element, resulting in a linear O(n) complexity. The process of removing an element in a ArrayList is similar to that of adding an element, in that it may require elements to be relocated. Removing the element at the end of the list simply requires the element at the final index to be removed from the array, without anything being moved. However, removing an element from the middle of the list requires the following elements be have their index reduced by 1 to fill the gap, also an O(n) complexity. Therefore, ArrayList implementation will incur a performance penalty over a LinkedList when attempting to add or remove elements at arbitrary locations.

A LinkedList uses pointers to indicate succeeding elements while storing them in an unordered fashion, as opposed to the contiguous nature of ArrayLists. This offers an advantage in performance when adding or removing elements anywhere in the List, except for at the end of the List. These operations should occur in a constant period (O(1)), as opposed to the linear period of an ArrayList implementation (O(n)) as elements do not need to be relocated during the operation, they only require the pointer’s value to be changed.

Random access is not a strong point of LinkedLists, as getting an element in a LinkedList requires the List to iterate through all elements until the desired element is found, giving it a performance of O(n), making it much less efficient than an ArrayList. Inserting an item in a LinkedList will have a constant complexity (O(1)), unless the item is at the end. To insert at the head, the pointer of the new element will point to the original root element, and the root pointer will point to the new element. Inserting in the middle of the List will require the pointer of the previous element to be copied to the new element’s pointer, and then the old pointer changed to point to the new element. Inserting at the end of the list requires the LinkedList to loop internally until a null pointer is found, this null pointer is then altered to point to the new element and the new elements pointer will remain null. Removing an element in a linked list requires the pointer of the element to be copied to the previous item in the list, so that nothing points to the element, and it is then erased.

The choice of ArrayList or LinkedList depends on the main operations that will be carried out on the contained elements. If the intended usage will rely heavily on insertion and removal of elements specifically at the end of the List, or a significant use of set and get methods, an ArrayList will offer greater efficiency due to its increased random access performance and contiguous nature of storage. Though, a List whose purpose will see more elements added and removed within the list, especially toward the front, will greatly benefit from the use of LinkedLists since elements will not have to be moved. LinkedLists also offer an advantage when data is growing at a significant rate, as an ArrayList will incur performance issues when the backing array reaches capacity and must copy its contents to a new array, a process not present in a LinkedList.

Computational complexity refers to the amount of computer resources required to perform an operation, the Big O notation is an example of measuring complexity. Big O usually measures the time or memory required during the execution of an algorithm, and can be used when deciding on the optimal algorithm for a task. For example, O(1) is considered fast as the operation may occur in almost instant time. An algorithm of complexity O(n), however, is proportional to the value of n and such will have a proportional increase in resources to complete

## Investigation

### Method

To get timing for methods, a large text file containing a list of 300000 randomly generated strings of random length, varying between 3 and 60 characters, was generated. This text file was then read into an ArrayList, on which several operations were performed. A call to System.nanoTime() was made before the operation and stored in variable, then another call made after the operation had complete and the difference between the two times would be calculated and output to the terminal showing the time taken in which to complete the operation.

The operations used were; fetching a string from a calculated index, inserting a string at a calculated index and deleting a string at a calculated index. 50 of each operation were recorded across five sets of ten calls, and the average of each set of ten calculated. The average of these five tests was then taken to get an average for all 50 operation calls.

This process was then repeated but having read the text file instead into a LinkedList, in order to retrieve timings for accessing the same data locations from the other List implementation so that the timings of these operations could be compared against the two implementations.

### Conclusion

The results of my investigation show that the ArrayList is significantly faster than the LinkedList at fetching data from the designated locations within the list, with the ArrayList averaging 729.08 nanoseconds and the LinkedList averaging 5444.1 nanoseconds. However, for both insertion and deletion of elements, the LinkedList was considerably faster, with the ArrayList taking an average of 159935 nanoseconds to insert and 146195.02 to delete. In comparison, the LinkedList took an average of 5741.02 nanoseconds to insert and 4931.02 nanoseconds to delete. While this makes ArrayList superior at fetching, it should be avoided unless this is the primary operation to be carried out on the List.

The range of time values returned by the operations carried out on the ArrayList is far greater than that of those returned by the LinkedList making it far less consistent where various operations are used. This suggests that, even if the fetch method is used as much as the insert and delete methods a LinkedList may still be favourable as the average time across all 150 operations for the ArrayList was 102286.3667 nanosecond compared to the LinkedList’s 5372.04 nanoseconds, making the ArrayList approximately 19 times slower overall throughout the testing process.

One thing that was apparent over the five sets of tests in each List, is that the first of the ten operations was considerably slower each time. This is because the first operation not only performs the intended function, it sets up the array indexes so that all other operations can be performed rather than the array merely acting as storage. This is apparent in both the LinkedList and ArrayLists, for example in the first test for the ArrayList, the first fetch operation took 2160 nanoseconds where all following fetch operations took either 270 or 540 nanoseconds. In the first test of the LinkedList implementation, the first fetch took 13772 nanoseconds where all following fetches took between 1350 and 5941 nanoseconds.

## References

Maciaszek, L. (2007). *Requirements analysis and system design*. 3rd ed. Harlow: Addison-Wesley, pp.338-340.

Xyzws.com. (2017). *Loose coupling and high cohesion*. [online] Available at: http://www.xyzws.com/scjp/SGS11/5/2 [Accessed 28 Nov. 2017].

Brockway, M. (2017). *Class Diagrams Coupling & Cohesion*.

Naftalin, M. and Wadler, P. (2007). *Java generics and collections*. 1st ed. Sebastopol, CA: O'Reilly, pp.213-222.

Dalton, N. (2017). *KF5008 Collections(2)*.

Techopedia.com. (2017). *What is Computational Complexity? - Definition from Techopedia*. [online] Available at: https://www.techopedia.com/definition/18466/computational-complexity [Accessed 30 Nov. 2017].

Black, P. and Pieterse, V. (2017). *big-O notation*. [online] Xlinux.nist.gov. Available at: https://xlinux.nist.gov/dads/HTML/bigOnotation.html [Accessed 30 Nov. 2017].