Applying Design by Contract to Java ArrayList Data Structure

Aldo Gabriele Di Rosa, Brenda Ruiz
& Jacopo Fidacaro

December 16, 2018

1 Selection Process

The selected project for implementing Design by Contract was the ArrayList and the Vector classes from Java Collections. We picked these two data structures given that their behaviour can be modelled by implementing Contracts.

2 Contract Implementation

First, we implemented an Interface ”ListContract” which defines the ArrayList methods and where the each of the Contracts were implemented. Then, we created a class ”ContractedArrayList” which implements our interface each of its methods. We created the analogous Interface and Class for the Vector, called VectorContract and ContractedVector respectively.

The approach followed for the implementation of the Contracts was first reading the class documentation for each method to understand and determine each one of the preconditions and postconditions. One of the challenges we faced during the implementation was understanding how to properly use the jSicko library. One obstacle we faced was understanding to name all variables with the same name so the library could parse them correctly. Also, in some cases, jSicko would produce errors which were difficult to understand what flaw they were pointing out and sometimes seemed arbitrary. (maybe add something more)

3 Contracts Overview

Overall, the preconditions we created require the parameters to be valid, for example, an index within the given size or the parameter was not null. For postconditions we ensured the appropiate changes were to the list and vector, such as changes in size and ensuring the list contains all elements after adding a Collection to it. The contracts we implemented for both classes were similar, given that they are both classes from Java Collections and have similar behavior. For the ArrayList class, a total of 16 Contracts and one Invariant were implemented (shown on Table 1). For the Vector class, 9 contracts were implemented (Table 2).

- Invariant: nonNegativeSize. Ensures the size of the list is always equal or greater than 0. This invariant was used for both classes.

4 Testing Outcomes

From our tests we were able to verify our preconditions worked properly. (?)

Method	**Precondition**	**Postcondition**
add(E e)	sizeIncreases	
add(int index, E element)	indexInRange	sizeIncreases
get(int index)	indexInRange	
remove(Object o)		sizeDoesNotIncrease
remove(int index)	indexInRange	sizeDecreases
addAll(Collection c)		containsAll
addAll(int index, Collection c)	indexInRange	
removeAll(Collection c)	collectionNotNull	
retainAll(Collection c)	collectionNotNull	
clear()	listEmpty	
set(int index, E element)	indexInRange	
indexOf(Object o)	equalObjectAtFirstOccurrence	
lastIndexOf(Object o)	equalObjectAtLastPosition	
subList(int fromIndex, int toIndex)	indexesInRange	
toArray(E[] a)	arrayNotNull	

Table 1: ArrayList Contracts

Method	**Precondition**	**Postcondition**
addElement(E obj)	sizeIncreases	
elementAt(int index)	indexWithinBounds	
removeElementAt(int index)	indexWithinBounds	
indexOf(Object o, int index)	nonNegativeIndex	sizeStaysTheSame
lastElement()	nonEmptyVector	
firstElement()	nonEmptyVector	
removeAllElements()	emptyVector	
removeIf(Predicate¡? super E¿ filter)	filterNotNull	
lastIndexOf(Object o, int index)	indexWithinBounds	sizeStaysTheSame

Table 2: Vector Contracts

5 Applicability to Other Projects

Naturally, the contracts we implemented could be applied for other collections, applying minor changes to fit each of their structures. They could also be useful for applications where these kinds of data structures are used.

6 Conclusions