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 is the ArrayList and the Vector classes from Java Util. We picked these two data structures given that their behaviour can be modelled by implementing Contracts and also because we are familiar with them. Initially, we thought about working with bachelor projects but decided not to work with them because they didn't have as many interfaces to work with. Then, we considered using a Java library such as Cloneable and Comparable but we realized that we first had to invest a lot of time getting familiar with it and trying to understand how to implement contracts to them. Therefore, we developed our project using ArrayList and Vector.

2 Project Description

In the end we decided to choose two data structures that implement the List interface such as ArrayList and Vector.

2.1 ArrayList

ArrayList is a resizable-array implementation that implements all optional list operations, and permits all elements, including null. Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The implementation of ArrayList is not synchronized, if multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally. Also this data structure implements the iterator interface and provides an iterator in order to access elements. In this project we applied the contracts in the following methods:

- toArray
- addAll
- removeAll
- retainAll
- clear
- get
- set
- add
- remove
- indexOf

- lastIndexOf
- subList

2.2 Vector

The Vector class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created. Also Vector provides an iterator, but the iterators returned by this class's iterator and listIterator methods are fail-fast: if the vector is structurally modified at any time after the iterator is created it will throw an exception. In this project we applied the contracts in the following methods:

- indexOf
- lastIndexOf
- elementAt
- firstElement
- lastElement
- removeElementAt
- removeAllElements
- addElement
- removeIf

3 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.

4 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 appropriate changes were made to the list and vector, such as changes in size and ensuring the list contained 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.

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 Testing Outcomes

From our tests we were able to verify our preconditions worked properly. In order to test the preconditions, we wrote some simple test classes with JUnit which declared some tests for our contracted `ArrayList` and `Vector`. Those methods would perform incorrect method calls that would trigger a precondition exception. This method only works for preconditions in case the contracted data structure is correct. In order to test the postconditions we would need to write an incorrect implementation of ArrayList that would brake the postconditions in very specific way (for example, in case a sort function would change the list size, a possible "sizeDidNotChange" postcondition would be broken by the implementation of the sort method). We will show and analyse one particular test case and display how we approached testing in general:

```java
@Test
  void testAddWithIndex() {
      ContractedArrayList<String> list = new ContractedArrayList<>();
      list.add("1");
      list.add("2");
      assertThrows(Contract.PreconditionViolation.class, () -> list.add(3, "1"));
  }
```

The test above checks for the `indexInRange` precondition on the `add` method of `ContractedArrayList` by creating a new list with two elements inside it and finally trying to use the add method that takes both an element to insert and an index to use for insertion. The last call though will break the `indexInRange` precondition, because the list has only two element and cannot accept any index referring to a index which is not available in the list (in our case, index 3). The assert statement just checks for a thrown `PreconditionViolation`.

The implementation of the precondition tested is as follows:

```
@Pure
default boolean indexInRange(int index) {
      return (size() > index) && (index >= 0);
}
```

The method just checks whether the passed index is greater than zero and smaller than the size of the list minus one.

Our general approach to testing was to at first design a meaningful precondition for either one ore more methods of a data structure, then annotating the methods that could benefit from such a precondition and finally writing a test that would instantiate the data structure with the needed operations to construct a meaningful use case for the contracted methods (like calling `add` multiple times on the list to initialize it properly) and calling the contracted method itself on the list to check for precondition violations. The JSicko library itself helped us in doing so but at the same time limited some of the possible contracts we could design. For example, we were not able to properly design a precondition for any method instantiating an iterator, because tests would not work properly for those cases.

6 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. In general, contracts are very useful to track down issues during API and data structures implementations, since they can directly address any implementation and/or usage issue. This can also be extended to complex interfaces in general, especially if they expose multiple methods to be used by external users, so that really need to have a very consistent behavior (that can be checked with postconditions) and at the same time be safe in case of misuse by the user (preconditions help avoiding unexpected errors in this case). Overall, contracts are a way of enhancing code checks that can be tackled in various different ways. For example, functional programming and in particular maybe monads can avoid unwanted behavior that can be caused by the usage of `null` values. Test driven development is an alternative to postconditions checks too. Design by contract is though a more clean and consistent way to approach the same issues.

7 Conclusions

To conclude, after working with Contracts we noticed their usefulness in several aspects. First, they document in a precise way what a method or class is supposed to do, allowing for better readability and allowing developers to gain deeper understanding on the behavior of the code. Secondly, they help eliminate redundant checks within the methods, such as null checks or other parameter validations. Thirdly, they serve as a way for dealing with abnormal cases, leading to safer and more effective exception handling, consequently improving reliability. Given the above, we can lastly conclude they are a powerful testing framework, given that they serve as a systematic approach to building bug free systems.