**Group Members:**

1. Clayton Mottley
2. Xavier Gitiaux
3. Anowarul Kabir

**Bag Implementation**

Our implementation of generic uses a HashMap as a representant, with the constraints that the values in the map can only be positive integers. The abstraction function transforms the representant into a multiset where the multiplicity of each element is the corresponding value in the HashMap. Note that we interpreted cardinality as the total number of elements (potentially duplicated) in the bag.

**public class** Bag<E> {  
  
 *// rep: map each object to the count of that object in this*  
 *// rep-inv: range of map contains only positive integers*  
 *// Example: A bag of 2 cats and a dog is map = { cat=2, dog=1 }*  
  
**private** Map<E, Integer> **map**;  
  
 **public boolean** repOK(){  
 **for**(Integer count: **map**.values()){  
 **if** (count <= 0 ) **return false**;  
 }  
 **return true**;  
 }  
  
 **public** Bag() {  
 **map** = **new** HashMap<E, Integer>();  
 }  
  
 *// add 1 occurrence of e to this*  
**public void** insert(E e) {  
 **if**(isIn(e)) {  
 **int** counterForE = **map**.get(e);  
 **map**.replace(e, ++counterForE);  
 }  
 **else**{  
 **map**.put(e, 1);  
 }  
 }  
  
 *// remove 1 occurrence of e from this*  
**public void** remove(E e) {  
 **int** countItem = **map**.get(e);  
 **if** (countItem == 1) {  
 **map**.remove(e);  
 }  
 **else**{  
 **map**.replace(e, countItem -1);  
 }  
 }  
  
 *// return true iff e is in this*  
**public boolean** isIn(E e) {  
 **return map**.containsKey(e);  
 }  
  
 *// return cardinality of this*  
**public int** size() {  
 **int** counter = 0;  
 **for**(Integer count: **map**.values()){  
 counter += count;  
 }  
 **return** counter;  
 }  
  
 *// if this is empty throw ISE*  
 *// else return arbitrary element of this*  
**public** E choose() {  
 **if** (**map**.size() == 0) **throw new** IllegalStateException(**"Bag.choose"**);  
 Object keys[]= **map**.keySet().toArray();  
 **return** (E) keys[0];  
 }  
  
 *// conveniently, the <E,Integer> map is exactly the abstract state*  
**public** String toString() { **return map**.toString(); }  
  
}

**Comparison of Abstract function (AF) and rep-inv between Bag and LiskovGenericSet (LGS):**

AF of LGS maps an item at most once.

AF(c\_LGS) = c\_LGS[0], c\_LGS[1], … … … c\_LGS[n-1]

where n is the number of items in the set.

Rep-inv:

if n==0, c\_LGS is empty and

if n>0, c\_LGS[i] != c\_LGS[j] where (i != j) and (0<=i,j<n) and

c\_LGS.isIn(e) follows by c\_LGS.insert(e) returns always true and

c\_LGS.isIn(e) follows by c\_LGS.remove(e) returns always false

where e is an item.

AF of Bag maps an items multiple times and keeps track of how many times an item is in the bag.

AF(c\_bag) = c\_bag[0], c\_bag[1], … … … c\_bag[n-1]

where c\_bag[i] = (itemKey, count)

where 0<=i<n and

n is the number of unique items in the bag and

count indicates how many times itemKey is in the bag

Rep-inv:

if n==0, c\_bag is empty and

if n>0, c\_bag[i].itemKey != c\_bag[j].itemKey where (i != j) and (0<=i,j<n) and

c\_bag.isIn(e) follows by c\_bag .insert(e) returns always true and

c\_bag[i].count > 0

where e is an item

**Properties of LiskovGenericSet (LGS):**

A generic Set is a mutable set of elements E with size equal or greater than 0 and no duplicates elements. Because there are no duplicates in Sets, if a Set contains an object and the object is removed, the Set will not contain the object. We wrote a JUnit theory to test this property for Sets – *remove()* followed by *isIn()* should always return false for Sets.

@DataPoints  
public static int[] *counts* = { 2, 100, 50, 23 };  
@DataPoints  
public static Object[] *objects* = { "string", 1, new Object() };

// Property - Sets do not contain duplicates  
// Theory - Regardless how many times an object is inserted into a Set,  
// when the same object is removed, it will not be present in the Set  
@Theory  
public void testSetPropertyForSets(Object object, int count) {  
 LiskovGenericSet<Object> set = new LiskovGenericSet();  
 assumeNotNull(object);

// Fill Set with arbitrary count of object  
 for (int i = 0; i < count; i++) {  
 set.insert(object);  
 }  
 assumeTrue(set.isIn(object));  
  
 set.remove(object);  
 assertFalse(set.isIn(object));  
}

To our expectation, this theory passed for Sets of any Object type and also regardless of how many times an object was added to the set – the object was always no longer an element of the Set after calling *remove()*.

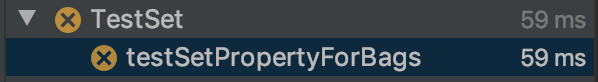


**Is Bag a legitimate subtype of LiskovGenericSet (LGS)?**

If Bags are subtypes of Sets they must adhere to the same properties as Sets as well – meaning Bags cannot contain duplicates. To test this, we applied the same Set property test to Bags.

// Property - Sets do not contain duplicates  
// If Bags are subtypes of Sets then they must pass this property test  
@Theory  
public void testSetPropertyForBags(Object object, int count) {  
 Bag<Object> bag = new Bag<Object>();  
 assumeNotNull(object);  
  
 // Fill Bag with arbitrary count of object  
 for (int i = 0; i < count; i++) {  
 bag.insert(object);  
 }  
 assumeTrue(bag.isIn(object));  
  
 bag.remove(object);  
 assertFalse(bag.isIn(object));  
}

This theory failed for Bag – if an element was inserted multiple times into a Bag it would still be present in the Bag after *remove(),* therefore Bag can contain duplicates. Because it fails the Set property test, it is not possible for Bag to be a valid subtype of Set based on the Property rule.



**Properties of Bag:**

A generic bag is a mutable multiset whose cardinality increases by one at each insertion and decreases by one at each removal. The size of a bag must equal or greater than 0.

To test this property, we wrote JUnit test that test whether cardinality gets incremented by one after inserting any element.

@Theory  
public void testBagPropertyForBags(Object object, **int** count) {  
 *assumeNotNull*(object);  
  
 Bag<Object> bag = **new** Bag<Object>();  
  
 *// Fill Set with arbitrary count of object*  
  
**for** (**int** i = 0; i < count; i++) {  
 bag.insert(object);  
 }  
  
 **int** size = bag.size();  
 bag.insert(*objects*);  
 *assertTrue*(bag.size() == size + 1);  
  
}

The test passes for Bag.

**Is LiskovGenericSet (LGS) a legitimate subtype of Bag?**

LiskovGenericSet violates the Bag property because after insertion of an element that is already in the LiskovGenericSet, the cardinality is not increased by one. Therefore, LiskovGenericSet is not a subtype of Bag. It is a violation of an evolution property. We can write a JUnit theory:

@Theory  
public void testBagPropertyForSets(Object object, **int** count) {  
 LiskovGenericSet<Object> set = **new** LiskovGenericSet<>();  
 *assumeNotNull*(object);  
  
 *// Fill Bag with arbitrary count of object*  
**for** (**int** i = 0; i < count; i++) {  
 set.insert(object);  
 }  
  
 *assumeTrue*(set.isIn(object));  
 **int** size = set.size();  
 set.insert(object);  
 *assertEquals*(set.size(), size + 1);  
}

The test fails for LiskovGenericSet.

**Contributions**: We first came over our own solutions. Then we discussed over the cardinality of the bag, Bag properties, Set properties and Junit theories. The story combines Xavier’s Bag implementation, Anowarul’s rep-inv and AF comparison and Clayton’s Junit theories. Finally, we answered two questions whether Bag is a legitimate subtype of LiskovGenericSet or vice versa.