**Projeto: Guava**

**Confiança Mínima: 0%**

1. **Erro sem estruturas de controle**

**Padrão:**

**Usuários que chamam:**

Table$Cell.getRowKey()

**Também chamam:**

Table$Cell.getRowKey()

Suporte: 0.000909 (6)

Confiança: 0.249

**Método Minerado:**

**public** Builder<R, C, V> put(Cell<? **extends** R, ? **extends** C, ? **extends** V> cell) {

**if** (cell **instanceof** Tables.ImmutableCell) {

*checkNotNull*(cell.getRowKey());

*checkNotNull*(cell.getColumnKey());

*checkNotNull*(cell.getValue());

@SuppressWarnings("unchecked") // all supported methods are covariant

Cell<R, C, V> immutableCell = (Cell<R, C, V>) cell;

cells.add(immutableCell);

} **else** {

put(cell.getRowKey(), cell.getColumnKey(), cell.getValue());

}

**return** **this**;

}

**Padrão não sendo utilizado:**

RegularImmutableTable.SparseImmutableTable(ImmutableList<Cell<R, C, V>> cellList,

ImmutableSet<R> rowSpace, ImmutableSet<C> columnSpace) {

Map<R, Integer> rowIndex = Maps.newHashMap();

Map<R, Map<C, V>> rows = Maps.newLinkedHashMap();

**for** (R row : rowSpace) {

rowIndex.put(row, rows.size());

rows.put(row, **new** LinkedHashMap<C, V>());

}

Map<C, Map<R, V>> columns = Maps.newLinkedHashMap();

**for** (C col : columnSpace) {

columns.put(col, **new** LinkedHashMap<R, V>());

}

**int**[] iterationOrderRow = **new** **int**[cellList.size()];

**int**[] iterationOrderColumn = **new** **int**[cellList.size()];

**for** (**int** i = 0; i < cellList.size(); i++) {

Cell<R, C, V> cell = cellList.get(i);

R rowKey = cell.getRowKey();

C columnKey = cell.getColumnKey();

V value = cell.getValue();

iterationOrderRow[i] = rowIndex.get(rowKey);

Map<C, V> thisRow = rows.get(rowKey);

iterationOrderColumn[i] = thisRow.size();

V oldValue = thisRow.put(columnKey, value);

**if** (oldValue != **null**) {

**throw** **new** IllegalArgumentException("Duplicate value for row=" + rowKey + ", column="

+ columnKey + ": " + value + ", " + oldValue);

}

columns.get(columnKey).put(rowKey, value);

}

**this**.iterationOrderRow = iterationOrderRow;

**this**.iterationOrderColumn = iterationOrderColumn;

ImmutableMap.Builder<R, Map<C, V>> rowBuilder = ImmutableMap.builder();

**for** (Map.Entry<R, Map<C, V>> row : rows.entrySet()) {

rowBuilder.put(row.getKey(), ImmutableMap.copyOf(row.getValue()));

}

**this**.rowMap = rowBuilder.build();

ImmutableMap.Builder<C, Map<R, V>> columnBuilder = ImmutableMap.builder();

**for** (Map.Entry<C, Map<R, V>> col : columns.entrySet()) {

columnBuilder.put(col.getKey(), ImmutableMap.copyOf(col.getValue()));

}

**this**.columnMap = columnBuilder.build();

}

1. **Erro sem estruturas de controle**

**Padrão:**

**Usuários que chamam:**

Table$Cell.getRowKey()

**Também chamam:**

Table$Cell.getRowKey()

Suporte: 0.000909 (6)

Confiança: 0.249

**Padrão não sendo utilizado:**

DenseImmutableTable(ImmutableList<Cell<R, C, V>> cellList,

ImmutableSet<R> rowSpace, ImmutableSet<C> columnSpace) {

@SuppressWarnings("unchecked")

V[][] array = (V[][]) **new** Object[rowSpace.size()][columnSpace.size()];

**this**.values = array;

**this**.rowKeyToIndex = makeIndex(rowSpace);

**this**.columnKeyToIndex = makeIndex(columnSpace);

rowCounts = **new** **int**[rowKeyToIndex.size()];

columnCounts = **new** **int**[columnKeyToIndex.size()];

**int**[] iterationOrderRow = **new** **int**[cellList.size()];

**int**[] iterationOrderColumn = **new** **int**[cellList.size()];

**for** (**int** i = 0; i < cellList.size(); i++) {

Cell<R, C, V> cell = cellList.get(i);

R rowKey = cell.getRowKey();

C columnKey = cell.getColumnKey();

**int** rowIndex = rowKeyToIndex.get(rowKey);

**int** columnIndex = columnKeyToIndex.get(columnKey);

V existingValue = values[rowIndex][columnIndex];

checkArgument(existingValue == **null**, "duplicate key: (%s, %s)", rowKey, columnKey);

values[rowIndex][columnIndex] = cell.getValue();

rowCounts[rowIndex]++;

columnCounts[columnIndex]++;

iterationOrderRow[i] = rowIndex;

iterationOrderColumn[i] = columnIndex;

}

**this**.iterationOrderRow = iterationOrderRow;

**this**.iterationOrderColumn = iterationOrderColumn;

**this**.rowMap = **new** RowMap();

**this**.columnMap = **new** ColumnMap();

}

**Padrão não sendo utilizado:**

**REFACTOR**

SparseImmutableTable.SparseImmutableTable(ImmutableList<Cell<R, C, V>> cellList,

ImmutableSet<R> rowSpace, ImmutableSet<C> columnSpace) {

Map<R, Integer> rowIndex = Maps.newHashMap();

Map<R, Map<C, V>> rows = Maps.newLinkedHashMap();

**for** (R row : rowSpace) {

rowIndex.put(row, rows.size());

rows.put(row, **new** LinkedHashMap<C, V>());

}

Map<C, Map<R, V>> columns = Maps.newLinkedHashMap();

**for** (C col : columnSpace) {

columns.put(col, **new** LinkedHashMap<R, V>());

}

**int**[] iterationOrderRow = **new** **int**[cellList.size()];

**int**[] iterationOrderColumn = **new** **int**[cellList.size()];

**for** (**int** i = 0; i < cellList.size(); i++) {

Cell<R, C, V> cell = cellList.get(i);

R rowKey = cell.getRowKey();

C columnKey = cell.getColumnKey();

V value = cell.getValue();

iterationOrderRow[i] = rowIndex.get(rowKey);

Map<C, V> thisRow = rows.get(rowKey);

iterationOrderColumn[i] = thisRow.size();

V oldValue = thisRow.put(columnKey, value);

**if** (oldValue != **null**) {

**throw** **new** IllegalArgumentException("Duplicate value for row=" + rowKey + ", column="

+ columnKey + ": " + value + ", " + oldValue);

}

columns.get(columnKey).put(rowKey, value);

}

**this**.iterationOrderRow = iterationOrderRow;

**this**.iterationOrderColumn = iterationOrderColumn;

ImmutableMap.Builder<R, Map<C, V>> rowBuilder = ImmutableMap.builder();

**for** (Map.Entry<R, Map<C, V>> row : rows.entrySet()) {

rowBuilder.put(row.getKey(), ImmutableMap.copyOf(row.getValue()));

}

**this**.rowMap = rowBuilder.build();

ImmutableMap.Builder<C, Map<R, V>> columnBuilder = ImmutableMap.builder();

**for** (Map.Entry<C, Map<R, V>> col : columns.entrySet()) {

columnBuilder.put(col.getKey(), ImmutableMap.copyOf(col.getValue()));

}

**this**.columnMap = columnBuilder.build();

}

1. **Erro sem estruturas de controle**

**Padrão:**

**Usuários que chamam:**

AbstractFuture.setException(java.lang.Throwable)

**Também chamam:**

AbstractFuture.setException(java.lang.Throwable)

AbstractFuture.setException(java.lang.Throwable)

Suporte: 0.0009909 (6)

Confiança: 0.75%

**Método Minerado:**

**public** **void** failed(State from, Throwable failure) {

**switch** (from) {

**case** *STARTING*:

startup.setException(failure);

shutdown.setException(**new** Exception("Service failed to start.", failure));

**break**;

**case** *RUNNING*:

shutdown.setException(**new** Exception("Service failed while running", failure));

**break**;

**case** *STOPPING*:

shutdown.setException(failure);

**break**;

**case** *TERMINATED*: /\* fall-through \*/

**case** *FAILED*: /\* fall-through \*/

**case** *NEW*: /\* fall-through \*/

**default**:

**throw** **new** AssertionError("Unexpected from state: " + from);

}

}

**Padrão não sendo utilizado:**

**public** **void** run() {

**try** {

delegate.run();

} **catch** (Throwable t) {

setException(t);

**throw** Throwables.propagate(t);

}

}

1. **Erro sem estruturas de controle**

**Padrão:**

**Usuários que chamam:**

TypeVariable.getBounds()

**Também chamam:**

WildcardType.getUpperBounds()

Suporte: 0.000909 (6)

Confiança: 0.75

**Método Minerado:**

**static** Type getComponentType(Type type) {

*checkNotNull*(type);

**if** (type **instanceof** Class) {

**return** ((Class<?>) type).getComponentType();

} **else** **if** (type **instanceof** GenericArrayType) {

**return** ((GenericArrayType) type).getGenericComponentType();

} **else** **if** (type **instanceof** WildcardType) {

**return** *subtypeOfComponentType*(((WildcardType) type).getUpperBounds());

} **else** **if** (type **instanceof** TypeVariable) {

**return** *subtypeOfComponentType*(((TypeVariable<?>) type).getBounds());

} **else** {

**return** **null**;

}

}

**private** **void** introspect(Type type) {

**if** (!introspectedTypes.add(type)) {

**return**;

}

**if** (type **instanceof** ParameterizedType) {

introspectParameterizedType((ParameterizedType) type);

} **else** **if** (type **instanceof** Class) {

introspectClass((Class<?>) type);

} **else** **if** (type **instanceof** TypeVariable) {

**for** (Type bound : ((TypeVariable<?>) type).getBounds()) {

introspect(bound);

}

} **else** **if** (type **instanceof** WildcardType) {

**for** (Type bound : ((WildcardType) type).getUpperBounds()) {

introspect(bound);

}

}

}

**Padrão não sendo utilizado:**

Type resolveTypeVariable(TypeVariable<?> var, TypeResolver guardedResolver) {

checkNotNull(guardedResolver);

Type type = map.get(var);

**if** (type == **null**) {

Type[] bounds = var.getBounds();

**if** (bounds.length == 0) {

**return** var;

}

**return** Types.newTypeVariable(

var.getGenericDeclaration(),

var.getName(),

guardedResolver.resolveTypes(bounds));

}

// in case the type is yet another type variable.

**return** guardedResolver.resolveType(type);

}

1. **Erro sem estruturas de controle**

**Padrão:**

**Usuários que chamam:**

TypeVariable.getBounds()

**Também chamam:**

WildcardType.getUpperBounds()

Suporte: 0.000909 (6)

Confiança: 0.75

**Padrão não sendo utilizado:**

Type resolveInternal(TypeVariable<?> var, TypeTable forDependants) {

Type type = map.get(var);

**if** (type == **null**) {

Type[] bounds = var.getBounds();

**if** (bounds.length == 0) {

**return** var;

}

**return** Types.newTypeVariable(

var.getGenericDeclaration(),

var.getName(),

**new** TypeResolver(forDependants).resolveTypes(bounds));

}

// in case the type is yet another type variable.

**return** **new** TypeResolver(forDependants).resolveType(type);

}

}

**Confiança Mínima: 20%**

1. **Erro com estruturas de controle**

**Padrão:**

**Usuários que chamam:**

java.util.Iterator.next()

java.util.Iterator.next()

**Também chamam:**

com.google.common.collect.Multiset.contains(java.lang.Object)

Suporte: 0.000909 (6)

Confiança: 20%

**Método Minerado (Supersequência minerada):**

**public** **static** **boolean** contains(Iterator<?> iterator, @Nullable Object element)

{

**if** (element == **null**) {

**while** (iterator.hasNext()) {

**if** (iterator.next() == **null**) {

**return** **true**;

}

}

} **else** {

**while** (iterator.hasNext()) {

**if** (element.equals(iterator.next())) {

**return** **true**;

}

}

}

**return** **false**;

}

**Padrão não sendo utilizado:**

**public** **void** testAsMapEntriesUpdate() {

multimap.put("foo", 1);

multimap.put("foo", 3);

Collection<Entry<String, Collection<Integer>>> entries =

multimap.asMap().entrySet();

Entry<String, Collection<Integer>> entry = entries.iterator().next();

Collection<Integer> values = entry.getValue();

multimap.put("foo", 5);

assertEquals(3, values.size());

assertTrue(values.contains(5));

values.add(7);

assertSize(4);

assertTrue(multimap.containsValue(7));

multimap.put("bar", 4);

assertEquals(2, entries.size());

assertSize(5);

assertTrue(entries.remove(entry));

assertSize(1);

assertFalse(multimap.containsKey("foo"));

assertTrue(multimap.containsKey("bar"));

assertFalse(entries.remove("foo"));

assertFalse(entries.remove(

Maps.immutableEntry("foo", Collections.singleton(2))));

assertSize(1);

Iterator<Entry<String, Collection<Integer>>> iterator =

entries.iterator();

assertTrue(iterator.hasNext());

iterator.next();

iterator.remove();

assertFalse(iterator.hasNext());

assertSize(0);

assertTrue(multimap.isEmpty());

multimap.put("bar", 8);

assertSize(1);

entries.clear();

assertSize(0);

}

1. **Erro com estruturas de controle**

**Padrão:**

**Usuários que chamam:**

java.util.Iterator.next()

java.util.Iterator.next()

**Também chamam:**

com.google.common.collect.Multiset.contains(java.lang.Object)

Suporte: 0.000909 (6)

Confiança: 20%

**Padrão não sendo utilizado:**

**public** **void** testEntrySetValue() {

multimap.put("foo", 1);

multimap.put("bar", 1);

Collection<Entry<String, Integer>> entries = multimap.entries();

Iterator<Entry<String, Integer>> iterator = entries.iterator();

Entry<String, Integer> entrya = iterator.next();

Entry<String, Integer> entryb = iterator.next();

**try** {

entrya.setValue(3);

fail();

} **catch** (UnsupportedOperationException expected) {}

assertTrue(multimap.containsEntry("foo", 1));

assertTrue(multimap.containsEntry("bar", 1));

assertFalse(multimap.containsEntry("foo", 2));

assertFalse(multimap.containsEntry("bar", 2));

assertEquals(1, (**int**) entrya.getValue());

assertEquals(1, (**int**) entryb.getValue());

}

1. **Erro com estruturas de controle**

**Padrão:**

**Usuários que chamam:**

java.util.Iterator.next()

java.util.Iterator.next()

**Também chamam:**

com.google.common.collect.Multiset$Entry.getElement()

Suporte: 0.000909 (6)

Confiança: 20%

**Método Minerado (Supersequência minerada):**

**public** **static** **boolean** contains(Iterator<?> iterator, @Nullable Object element)

{

**if** (element == **null**) {

**while** (iterator.hasNext()) {

**if** (iterator.next() == **null**) {

**return** **true**;

}

}

} **else** {

**while** (iterator.hasNext()) {

**if** (element.equals(iterator.next())) {

**return** **true**;

}

}

}

**return** **false**;

}

**Padrão não sendo utilizado:**

**public** **void** testAsMapEntriesUpdate() {

multimap.put("foo", 1);

multimap.put("foo", 3);

Collection<Entry<String, Collection<Integer>>> entries =

multimap.asMap().entrySet();

Entry<String, Collection<Integer>> entry = entries.iterator().next();

Collection<Integer> values = entry.getValue();

multimap.put("foo", 5);

assertEquals(3, values.size());

assertTrue(values.contains(5));

values.add(7);

assertSize(4);

assertTrue(multimap.containsValue(7));

multimap.put("bar", 4);

assertEquals(2, entries.size());

assertSize(5);

assertTrue(entries.remove(entry));

assertSize(1);

assertFalse(multimap.containsKey("foo"));

assertTrue(multimap.containsKey("bar"));

assertFalse(entries.remove("foo"));

assertFalse(entries.remove(

Maps.immutableEntry("foo", Collections.singleton(2))));

assertSize(1);

Iterator<Entry<String, Collection<Integer>>> iterator =

entries.iterator();

assertTrue(iterator.hasNext());

iterator.next();

iterator.remove();

assertFalse(iterator.hasNext());

assertSize(0);

assertTrue(multimap.isEmpty());

multimap.put("bar", 8);

assertSize(1);

entries.clear();

assertSize(0);

}

1. **Erro com estruturas de controle**

**Padrão:**

**Usuários que chamam:**

java.util.Iterator.next()

java.util.Iterator.next()

**Também chamam:**

com.google.common.collect.Multiset$Entry.getElement()

Suporte: 0.000909 (6)

Confiança: 20%

**Padrão não sendo utilizado:**

**public** **void** testEntrySetValue() {

multimap.put("foo", 1);

multimap.put("bar", 1);

Collection<Entry<String, Integer>> entries = multimap.entries();

Iterator<Entry<String, Integer>> iterator = entries.iterator();

Entry<String, Integer> entrya = iterator.next();

Entry<String, Integer> entryb = iterator.next();

**try** {

entrya.setValue(3);

fail();

} **catch** (UnsupportedOperationException expected) {}

assertTrue(multimap.containsEntry("foo", 1));

assertTrue(multimap.containsEntry("bar", 1));

assertFalse(multimap.containsEntry("foo", 2));

assertFalse(multimap.containsEntry("bar", 2));

assertEquals(1, (**int**) entrya.getValue());

assertEquals(1, (**int**) entryb.getValue());

}

1. **Erro com estruturas de controle**

**Padrão:**

**Usuários que chamam:**

java.util.Iterator.next()

java.util.Iterator.next()

**Também chamam:** com.google.common.collect.Multisets.immutableEntry(com.google.common.collect.Multisets$(Lcom/google/common/collect/Multiset;Lcom/google/common/collect/Multiset;)Lcom/google/common/collect/Multiset;$E, I)

Suporte: 0.000909 (6)

Confiança: 20%

**Método Minerado (Supersequência minerada):**

**public** **static** **boolean** contains(Iterator<?> iterator, @Nullable Object element)

{

**if** (element == **null**) {

**while** (iterator.hasNext()) {

**if** (iterator.next() == **null**) {

**return** **true**;

}

}

} **else** {

**while** (iterator.hasNext()) {

**if** (element.equals(iterator.next())) {

**return** **true**;

}

}

}

**return** **false**;

}

**Padrão não sendo utilizado:**

**public** **void** testAsMapEntriesUpdate() {

multimap.put("foo", 1);

multimap.put("foo", 3);

Collection<Entry<String, Collection<Integer>>> entries =

multimap.asMap().entrySet();

Entry<String, Collection<Integer>> entry = entries.iterator().next();

Collection<Integer> values = entry.getValue();

multimap.put("foo", 5);

assertEquals(3, values.size());

assertTrue(values.contains(5));

values.add(7);

assertSize(4);

assertTrue(multimap.containsValue(7));

multimap.put("bar", 4);

assertEquals(2, entries.size());

assertSize(5);

assertTrue(entries.remove(entry));

assertSize(1);

assertFalse(multimap.containsKey("foo"));

assertTrue(multimap.containsKey("bar"));

assertFalse(entries.remove("foo"));

assertFalse(entries.remove(

Maps.immutableEntry("foo", Collections.singleton(2))));

assertSize(1);

Iterator<Entry<String, Collection<Integer>>> iterator =

entries.iterator();

assertTrue(iterator.hasNext());

iterator.next();

iterator.remove();

assertFalse(iterator.hasNext());

assertSize(0);

assertTrue(multimap.isEmpty());

multimap.put("bar", 8);

assertSize(1);

entries.clear();

assertSize(0);

}

1. **Erro com estruturas de controle**

**Padrão:**

**Usuários que chamam:**

java.util.Iterator.next()

java.util.Iterator.next()

**Também chamam:** com.google.common.collect.Multisets.immutableEntry(com.google.common.collect.Multisets$(Lcom/google/common/collect/Multiset;Lcom/google/common/collect/Multiset;)Lcom/google/common/collect/Multiset;$E, I)

Suporte: 0.000909 (6)

Confiança: 20%

**Padrão não sendo utilizado:**

**public** **void** testEntrySetValue() {

multimap.put("foo", 1);

multimap.put("bar", 1);

Collection<Entry<String, Integer>> entries = multimap.entries();

Iterator<Entry<String, Integer>> iterator = entries.iterator();

Entry<String, Integer> entrya = iterator.next();

Entry<String, Integer> entryb = iterator.next();

**try** {

entrya.setValue(3);

fail();

} **catch** (UnsupportedOperationException expected) {}

assertTrue(multimap.containsEntry("foo", 1));

assertTrue(multimap.containsEntry("bar", 1));

assertFalse(multimap.containsEntry("foo", 2));

assertFalse(multimap.containsEntry("bar", 2));

assertEquals(1, (**int**) entrya.getValue());

assertEquals(1, (**int**) entryb.getValue());

}

1. **Erro com estruturas de controle**

**Padrão:**

**Usuários que chamam:**

java.util.Iterator.next()

java.util.Iterator.next()

**Também chamam:**

com.google.common.collect.Multiset$Entry.getCount()

Suporte: 0.000909 (6)

Confiança: 20%

**Método Minerado (Supersequência minerada):**

**public** **static** **boolean** contains(Iterator<?> iterator, @Nullable Object element)

{

**if** (element == **null**) {

**while** (iterator.hasNext()) {

**if** (iterator.next() == **null**) {

**return** **true**;

}

}

} **else** {

**while** (iterator.hasNext()) {

**if** (element.equals(iterator.next())) {

**return** **true**;

}

}

}

**return** **false**;

}

**Padrão não sendo utilizado:**

**public** **void** testAsMapEntriesUpdate() {

multimap.put("foo", 1);

multimap.put("foo", 3);

Collection<Entry<String, Collection<Integer>>> entries =

multimap.asMap().entrySet();

Entry<String, Collection<Integer>> entry = entries.iterator().next();

Collection<Integer> values = entry.getValue();

multimap.put("foo", 5);

assertEquals(3, values.size());

assertTrue(values.contains(5));

values.add(7);

assertSize(4);

assertTrue(multimap.containsValue(7));

multimap.put("bar", 4);

assertEquals(2, entries.size());

assertSize(5);

assertTrue(entries.remove(entry));

assertSize(1);

assertFalse(multimap.containsKey("foo"));

assertTrue(multimap.containsKey("bar"));

assertFalse(entries.remove("foo"));

assertFalse(entries.remove(

Maps.immutableEntry("foo", Collections.singleton(2))));

assertSize(1);

Iterator<Entry<String, Collection<Integer>>> iterator =

entries.iterator();

assertTrue(iterator.hasNext());

iterator.next();

iterator.remove();

assertFalse(iterator.hasNext());

assertSize(0);

assertTrue(multimap.isEmpty());

multimap.put("bar", 8);

assertSize(1);

entries.clear();

assertSize(0);

}

1. **Erro com estruturas de controle**

**Padrão:**

**Usuários que chamam:**

java.util.Iterator.next()

java.util.Iterator.next()

**Também chamam:**

com.google.common.collect.Multiset$Entry.getCount()

Suporte: 0.000909 (6)

Confiança: 20%

**Padrão não sendo utilizado:**

**public** **void** testEntrySetValue() {

multimap.put("foo", 1);

multimap.put("bar", 1);

Collection<Entry<String, Integer>> entries = multimap.entries();

Iterator<Entry<String, Integer>> iterator = entries.iterator();

Entry<String, Integer> entrya = iterator.next();

Entry<String, Integer> entryb = iterator.next();

**try** {

entrya.setValue(3);

fail();

} **catch** (UnsupportedOperationException expected) {}

assertTrue(multimap.containsEntry("foo", 1));

assertTrue(multimap.containsEntry("bar", 1));

assertFalse(multimap.containsEntry("foo", 2));

assertFalse(multimap.containsEntry("bar", 2));

assertEquals(1, (**int**) entrya.getValue());

assertEquals(1, (**int**) entryb.getValue());

}