1 Solution for src/edu/uwm/cs351/ArrayPartCollection.java

`extends AbstractCollection<Part> implements Robot, Cloneable`

1.1 wellFormed

```
// 1. The "functions" and "parts" arrays must not be null.
if (functions == null || parts == null) {
            return report("Arrays must not be null.");
}
// 2. The functions and parts arrays are always the same length.
// Note: This makes sure that ensureCapacity (or any sort of dynamic array implementation)
// implemented correctly.
if (functions.length != parts.length) {
    return report("Arrays are not the same length.");
}

// 3. The size cannot be negative or greater than the length of the arrays.
if(size < 0) return report("size cannot be negative! size = " + size);
if(size > parts.length) return report("size is greater than the length of the arrays! arra

// 4. None of the first ”size” elements of either array can be null. (i.e., no holes)
for(int i = 0; i < size; ++i) {
    if(parts[i] == null) return report("parts[" + i + "] is null!");
    if(functions[i] == null) return report("functions[" + i + "] is null!");
}
```

1.2 Body of class

```
@Override //required
public boolean addPart(String function, Part part) {
    // Similar to HW 1 but with ensure capacity
    if(function == null || part == null)
        throw new NullPointerException("cannot add null function or part");
    assert wellFormed(): "invariant broke on entry to add";
    ensureCapacity(size + 1);
    functions[size] = function;
    parts[size] = part;
    ++size;
    ++version;
    assert wellFormed(): "invariant broke after add";
    return true;
}
```

```
@Override //required
public Part removePart(String function) {
    assert wellFormed(): "invariant broke at the beginning of removePart";
    for (int i=0; i < size; ++i) {
        if (function == null || function.equals(functions[i])) {
            Part p = parts[i];
            for(int j = i; j+1 < size; ++j)
            {
                functions[j] = functions[j+1];
                parts[j] = parts[j+1];
            }
            --size;
            ++version;
            functions[size] = null;
            parts[size] = null;
            return p;
        }
    }
    assert wellFormed(): "invariant broke at the end of remove";
    return null;
}


@Override //required
public Part getPart(String function, int index) {
    assert wellFormed(): "invariant broke at the beginning of getPart";
    if(index < 0) throw new IllegalArgumentException("index must be in range");
    if(index >= size) return null;
    if (function != null) {
        Iterator<Part> it = iterator(function);
        while (it.hasNext()) {
            Part p = it.next();
            if(index-- == 0) return p;
        }
    }
    else
    {
        return parts[index];
    }
    return null;
}


@Override // decorate
public ArrayPartCollection clone() {
    assert wellFormed() : "invariant broken in clone";
    ArrayPartCollection result;
    try {
        result = (ArrayPartCollection) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new AssertionError("forgot to implement cloneable?");
    }
    result.functions = functions.clone();
    result.parts = parts.clone();
    assert result.wellFormed() : "invariant broken in result of clone";
    assert wellFormed() : "invariant broken by clone";
    return result;
}


@Override // required
public Iterator<Part> iterator() {
    return new MyIterator(null);
}
```

```
/**
 * Return an iterator of the parts of the given function.
 * @param function function, null to mean all parts
 * @return iterator over those parts
 */
public Iterator<Part> iterator(String function) {
    return new MyIterator(function);
}

@Override // required
public int size() {
    return this.size;
}

@Override // efficiency
public void clear() {
    if (size == 0) return;
    ++version;
    size = 0;
}
```

```
implements Iterator<Part>
```

1.3 Iterator wellFormed

```
if(!ArrayPartCollection.this.wellFormed()) return false;
if(version != colVersion) return true;

//check if index is actually in the list
if(cur < 0 || cur > size)
    return report("cur is out of bounds");
if(next < 0 || next > size)
    return report("next is out of bounds");

if (cur != next) {
    if(cur > next)
        return report("cur must be before (or equal to) next");
    if (cur == size) {
        return report("can remove but nothing to remove");
    }
    if(function == null) {
        if(next != cur + 1)
            return report("function is null, next should be next element after cur");
    }
    else {
        for(int i = cur + 1; i < size; ++i) {
            if(function == functions[i] && next != i)
                return report("next should be pointing to the next element with functi
            if(i == next) break;
        }
        if (!function.equals(functions[cur])) {
            return report("element to remove has wrong function");
        }
    }
}
if (next < size && function != null) {
    if (!function.equals(functions[next])) {
        return report("function of next element is incorrect");
    }
}
```

1.4 Iterator constructor

```
    this.colVersion = version;
    this.function = func;
    this.next = 0;
    moveToMatch();
    this.cur = this.next;
```

1.5 Body of iterator

```
void moveToMatch() {
    while (function != null && next < size && !function.equals(functions[next])) {
        ++next;
    }
}

@Override //required
public boolean hasNext() {
    assert wellFormed(): "invariant broke in hasNext";
    checkVersion();
    return next < size;
}

private void checkVersion() {
    if(version != colVersion) throw new ConcurrentModificationException("bad version");
}

@Override //required
public Part next() {
    if(hasNext() == false) throw new NoSuchElementException("no more parts"); //also check
    assert wellFormed(): "invariant broke at the beginning of next";

    cur = next;
    ++next;
    moveToMatch();
    assert wellFormed(): "invariant broke at the end of next";
    return parts[cur];
}

@Override //Implementation
public void remove() {
    assert wellFormed(): "invariant broke at the beginning of remove";
    checkVersion();
    if (cur == next) throw new IllegalStateException("nothing to remove");

    for(int i = cur; i+1 < size; ++i)
    {
        functions[i] = functions[i+1];
        parts[i] = parts[i+1];
    }
    cur = --next;
    --size;
    functions[size] = null;
    parts[size] = null;
    colVersion = ++version;
    assert wellFormed(): "invariant broke at the end of remove";
}
```