Project #2
Specifications, Implementations, and Test Cases (Oh My!)

1 Objectives

There are three objectives to this project. First, you will gain experience developing a formal interface specification based on an abstract model. Second, you will gain experience implementing a class based solely on your understanding of its interface specification. Finally, you will gain additional experience implementing specification-based unit tests using the JUnit testing framework.

This is a short project, but you are encouraged to start early in case you run into unexpected problems with the mathematics.

2 Requirements

This project is divided into three components.

- **Specification.** For the first component of the project, you must develop the full formal specification of the SList interface included at the end of this document. This means that you should provide the appropriate pre-conditions, post-conditions, and preservation clauses. You should use the abstract model provided and base each method contract on the method’s informal description. Be sure to use the notation presented in class. (Note: You do not have to specify the behavior of abstractToString().) Note that SList is modeled as a pair of string of object. The model declaration gives each element a name – left and right, respectively. In a pre-condition or post-condition, you may use self.left to refer to the “left” string, and self.right to refer to the “right” string. If you use self alone, it refers to the pair of strings.

- **Implementation.** For the second component of the project, you must provide an implementation of the SList interface. Your class should be named SListImp1.

- **Testing.** For the third component of the project, you must develop specification-based unit tests to validate the correctness of your SList implementation. More precisely, you are required to implement a JUnit test class, TestSListImp1, that tests the correctness of SListImp1. As you did in the closed lab, you must provide one testXXX() method for each method defined by SList. For each testXXX() method, you must provide at least three test cases per method contract, taking care to test any “boundary cases”. Try your best to test the class thoroughly — include those cases that are most likely to exercise all paths through an implementation of SList.

Be sure to eliminate redundancy in your test suite by providing an appropriate implementation of setUp().
3 Submission Instructions

This project is due in class on March 17th. Absolutely no late assignments will be accepted. Please start early.

When your project is complete, archive your solution and use the handin command to submit your work. The command for this project is: handin.215.your_section 2 <filename>. You may use any standard archive format you like. Be sure to bring a hard copy of your solution to class.

4 Grading

Your project will be graded based on your adherence to the specified requirements, and the specification and programming guidelines discussed in class.

This is an intermediate course in software development. Your specification must be typed. Your source materials should be properly documented. Your source must compile. Your application must not crash. A violation of any of these requirements will result in an automatic zero. Test your application thoroughly.

Note that due to the small scale of this project, it will be weighted approximately 25% less than the first project.

5 Collaboration

You must work independently on this project. You must not discuss the problem or the solution with classmates. Any form of collaboration will be considered academic misconduct.

```java
public interface SLList {
    // modeled by: (left: string of object, right: string of object)
    // initial value: (<>,<>)

    void clear();
    // * This method may be called at any time.
    // * This method clears all entries from the left and right strings.

    void addRight(Object x);
    // * This method may be called at any time.
    // * This method prepends x to the right string.
    // * X is not modified by the method.

    Object removeRight();
    // * This method may be called when the right string is non-empty.
    // * This method removes and returns the leftmost entry from the right string.
}
```
Object getElementAt(int pos);
// * This method may be called when pos is greater
// than or equal to zero and less than the number of
// entries in the right string.
// * This method returns (but does not remove) the
// entry at position pos within the right string.
// * pos is not modified by the method.
// * self is not modified by the method.

void advance();
// * This method may be called when the right string
// is non-empty.
// * This method removes the leftmost entry from the
// right string and appends it to the left string.

void moveToStart();
// * This method may be called at any time.
// * This method first prepends the left string to the
// right string, and then sets the left string to
// empty-string.

void moveToFinish();
// * This method may be called at any time.
// * This method first appends the right string to the
// left string, and then sets the right string to
// empty-string.

int getLeftLength();
// * This method may be called at any time.
// * This method returns the length of the left
// string.
// * self is not modified by the method.

int getRightLength();
// * This method may be called at any time.
// * This method returns the length of the right
// string.
// * self is not modified by the method.

String abstractToString();
// * This method may be called at any time.
// * The String returned by this method should encode
// the abstract value of self.
// * The String returned by this method should follow
// the following pattern exactly:
// - (<>,<>)
// - (<1>,<>)
// - (<1,2>,<>)
// - (<1,2>,<1>)
// - (<1,2>,<1,2>)
// - ...
}

Listing 1: Informal Specification of SList