Survey Questionnaire on Software Change Contracts

Thanks you for participating in this survey on software change contracts. This survey is part of ongoing effort to improve software quality and your help is much appreciated.

The survey consists of two parts. In this first part of the survey, you are going to be asked five questions, and in the next part, three questions. In this first part, you will see the questions based on simple programs specially designed for this survey. And, in the next part, you will see similar questions based on a real program called AspectJ.

To help you understand the questions, we provide a sample question prior to actual questions. (This sample question is identical to the one distributed through IVLE.) We also ask 7 background questions to know your general familiarity with programming.

Matric Number: _____

Background Questions

1. Which year are you in, and what is your major?

   _____, _____

2. Rate your knowledge about Java language.
   (a) Never used it.
   (b) Beginner (e.g. have taken an introductory course)
   (c) Medium (e.g. have done some small projects with Java)
   (d) Proficient (e.g. have experience in developing real-life programs with Java)

3. What programming language are you most skillful at?

   _____

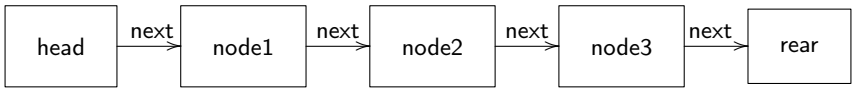
4. Rate your knowledge about the language you answered above if it is different from Java.
   (a) Never used it.
   (b) Beginner (e.g. have taken an introductory course)
   (c) Medium (e.g. have done some small projects with it)
   (d) Proficient (e.g. have experience in developing real-life programs with it)

5. Select the ways you specify your program (multiple answers possible).
   (a) I write comments that explain my program.
   (b) I write assert statements to express my assumption.
   (c) I write formal specification.

6. Rate your knowledge about program contract.
   (a) Never heard of it.
   (b) Heard of it, but has not used it.
   (c) Have written some program contracts.

7. Rate your knowledge about JML (Java Modeling Language).
   (a) Never heard of it before this course.
   (b) Heard of it, but has not used it.
   (c) Have used it.

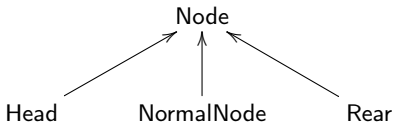
Sample Question. Consider the following linked list where the head and the rear of the list are distinguished from the rest of the list.



The head and rear are instances of class Head and Rear, respectively. That is,

Head head = new Head(); Rear rear = new Rear();

Meanwhile, nodes in the middle are instances of class NormalNode. All three classes, i.e., Head, Rear and NormalNode, are subclasses of Node. That is, the following is the class hierarchy for them:



Only NormalNode has a value field of the integer type as shown in the following:

```
public class NormalNode extends Node {
  Node next; // points to the next node and is not null
  int value;

  public boolean hasConsecutiveZeros() {
    if (value == 0) {
      if (((NormalNode) next).value == 0) { // may throw ClassCastException
        return true;
      }
    }
    return next.hasConsecutiveZeros();
  }
  /* the rest of the code is omitted */
}
```

We are interested in whether or not two consecutive nodes contain zeros, and the hasConsecutiveZeros method shown in the above answers to that question. For example, if node1 and node2 of the above figure have zeros as their values (i.e., node1.value == 0 and node2.value == 0), then node1.hasConsecutiveZeros() returns true.

However, the above hasConsecutiveZeros method has a bug. For example, if node3.hasConsecutiveZeros() is called for node3 of the above figure, ClassCastException is thrown because node3.next is cast to NormalNode despite that node3.next is an instance of Rear.

Now, write a change contract that expresses that the observed ClassCastException should not be thrown from hasConsecutiveZeros(). If necessary, use "next instanceof NormalNode" or similar instanceof expressions in the change contract.

```
signaled (ClassCastException) next instanceof NormalNode;
signals (ClassCastException) false;
```

(Continued)

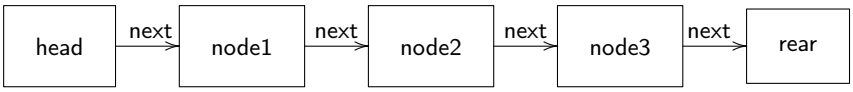
Now, fill in the blank in the below with a modified statement that respects the given change contract.

```
public class NormalNode extends Node {
  Node next; // points to the next node and is not null
  int value;

  public boolean hasConsecutiveZeros() {
    if (value == 0) {
      if ( next instanceof NormalNode ) {
        if (((NormalNode) next).value == 0) {
          return true;
        }
      } else {
        return false;
      }
    }
    return next.hasConsecutiveZeros();
  }
  /* the rest of the code is omitted */
}
```

Part I

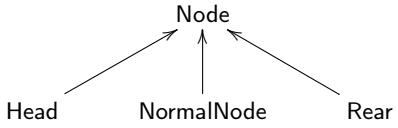
1. **[The linked list used in this question is the same as the one used in the sample question.]** Consider the following linked list where the head and the rear of the list are distinguished from the rest of the list.



The head and rear are instances of class Head and Rear, respectively. That is,

Head head = new Head(); Rear rear = new Rear();

Meanwhile, nodes in the middle are instances of class NormalNode. All three classes, i.e., Head, Rear and NormalNode, are subclasses of Node. That is, the following is the class hierarchy for them:



Only NormalNode has a value field of the integer type as shown in the following:

```
public class NormalNode extends Node {
  Node next; // points to the next node and is not null
  int value;

  public boolean hasConsecutiveZeros() {
    if (value == 0) {
      if (((NormalNode) next).value == 0) { // may throw ClassCastException
        return true;
      }
    }
    return next.hasConsecutiveZeros();
  }
  /* the rest of the code is omitted */
}
```

We are interested in whether or not two consecutive nodes of a linked list contain zeros, and the hasConsecutiveZeros method shown in the above answers to that question. For example, if node1 and node2 of the above figure have zeros as their values (i.e., node1.value == 0 and node2.value == 0), then node1.hasConsecutiveZeros() returns true.

However, the above hasConsecutiveZeros method has a bug. For example, if node3.hasConsecutiveZeros() is called for node3 of the above figure, ClassCastException is thrown because node3.next is cast to NormalNode despite that node3.next is an instance of Rear.

Q. Suppose that we now want to throw a NonNormalNodeException instead of a ClassCastException from hasConsecutiveZeros(). Write a change contract accordingly. If necessary, use "next instanceof Rear" or similar instanceof expressions in the change contract.

signaled (ClassCastException) next instanceof Rear;
signals (NormalNodeException) true;

2. Consider the linked list used in the previous question again. We now want to add an additional method tailList() to class Node. This new tailList() method is expected to return a list consisting of the nodes in the tail. Taking the figure used in the previous question as an example, node1.tailList() should return a list consisting of node2, node3, and rear.

   Each subclass of Node, i.e., Head, NormalNode and Rear, should override the tailList() method. For example, the following shows the tailList() of NormalNode.

```
public class NormalNode extends Node {
  private Node next; // not null
  private int value;

  public List tailList() {
    List list = new List(); // make a fresh list
    Head head = new Head(); // make a fresh head
    list.head = head; // set the head
    head.next = this.next; // the new list starts with the next node
    return list;
  }
  /* the rest of the code is omitted */
}
```

   Similarly, tailList() is overridden in Rear as well:

```
public class Rear extends Node {
  public List tailList() {
    return null;
  }
  /* the rest of the code is omitted */
}
```

   However, the above tailList() of Rear turns out to be buggy causing NullPointerException. So, we wrote a change contract as follows:

```
ensured \result == null;
ensures (\result instanceof List) && (\result.isEmpty() == true);
```

   Note that class List has method isEmpty() that returns true if the current List instance represents an empty list. Also note that an empty list is constructed by calling "new List()".

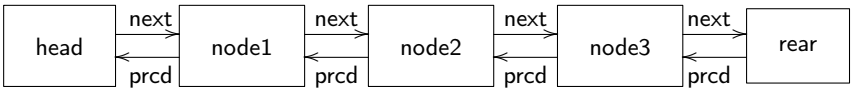
   Q1. Now, explain in English what the above change contract means:

   When given the input that results in returning null in the previous version of tailList, the new version of tailList returns a value v that satisfiles the following: (1) v is an instance of List and (2) v.isEmpty() is true. (Also, when given the rest of the input, two versions return the same value.)

   Q2. Fill in the following blank with a modified statement that respects the given change contract.

```
public class Rear extends Node {
  public List tailList() {
    return new List( );
  }
}
```

3. We are now going to extend the previous linked list to a doubly linked list like the following.



Classes should be extended and modified accordingly. For example, the following shows that class NormalNode now contains an extra field prcd to point to the preceding node.

```
public class NormalNode extends Node {
  private Node prcd; // points to the preceding node.
  private Node next;
  private int value;

  public boolean hasConsecutiveZeros(boolean forward) {
    // should extend it
  }
  /* the rest of the code is omitted */
}
```

The above also shows that method hasConsecutiveZeros now has a parameter forward. Depending on its boolean value, the direction to search for zeros are determined. While in the previous version zeros are searched for only in the forward direction, we now expect the extended hasConsecutiveZeros to be able to search for zeros in both directions.

Part of the above extension to a doubly linked list can be automated by following a few refactoring steps. After applying refactoring steps of adding a field and adding a parameter, we get the following change contract template for method hasConsecutiveZeros.

Q1. We ask you to fill in the blank. Note that the following change contract should say that only if forward is false, hasConsecutiveZeros may behave differently from before, and otherwise the same behavior should be preserved.

```
new_field prcd:Node;
new_param forward:boolean;
matches prcd == null && forward == true;
```

Q2. Also, explain in English what the above change contract means:

(1) The new version hasConsecuriveZeros can access to a new field prcd of Node type and a new parameter forward of boolean type. (2) Its behavior should be preserved if prcd is null and forward is true. (Also, (3) its behavior may change if the matches condition does not hold.)

4. The following shows a class that implements Iterator. Any Iterator class must have a next method that returns the next item to iterate over. The next method in the below returns either null if there is no more item to iterate over or a non-null value otherwise (i.e., items.get(currentIndex)).

```
import java.util.NoSuchElementException;

public class CustomIterator implements Iterator {
  private int currentIndex, size;
  private NonNullList items; // a list with no null item

  public Object next() {
    if (currentIndex < size) {
      Object result = items.get(currentIndex);
      currentIndex++;
      return result; // return a non-null value
    } else {
      return null;
    }
  }
  /* the rest of the code is omitted */
}
```

Now, we want to modify the above next method according to the following change contract.

```
ensured \result == null;
signals (NoSuchElementException) true;
```

Q1. Explain in English what the above change contract means:

When given the input that results in returning null in the previous version of next, the new version of next throws an exception of type NoSuchElementException.

(Continued in the next page)

Q2. Fill in the blank in the below with a modified statement that respects the given change contract. You can use the following API if necessary.

– NoSuchElementException() of class NoSuchElementException:
  • This is the default constructor of class NoSuchElementException.

```
public class CustomIterator implements Iterator {
  private int currentIndex, size;
  private NonNullList items; // a list with no null item

  public Object next() {
    if (currentIndex < size) {
      Object result = items.get(currentIndex);
      currentIndex++;
      return result; // return a non-null value
    } else {
      throw new NoSuchElementException( );
    }
  }
}
```

5. The following shows the Person class that holds information about the first name, the last name, and so on. We assume that none of these strings is null.

```
public class Person {
  private String firstName; // non-null
  private String lastName; // non-null
  private Nationality nationality; // non-null

  public boolean hasSameName(String first, String last) {
    return firstName.equals(first) && lastName.equals(last);
  }

  public String getFirstName() { return this.firstName; }
  public String getLastName() { return this.lastName; }
  /* the rest of the code is omitted */
}
```

The above class has a boolean method hasSameName that returns true if given two parameters first and last match the fields firstName and lastName, respectively. We assume that those two parameters, first and last, cannot be null.

Now, we want to shorten the parameter list of hasSameName as follows. Again, we assume that the person parameter cannot be null.

```
public boolean hasSameName(Person person) {
    return person.getFirstName().equals(firstName)
        && person.getLastName().equals(lastName);
}
```

When we shorten the parameter list, an accompanying tool generated the following change contract template:

```
old_param first:String, last:String;
new_param person:Person;
matches
```

person.getFirsName().equals(\prev(first)) &&
person.getLastName().equals(\prev(last));

Q1. Fill in the above blank.

Q2. Also, explain in English what the above change contract means:

(1) The new version hasSameName does not use its two old parameters, first and last. Instead, it has a new parameter, person of type Person. (2) Its behavior should be preserved if the conditions of the matches clause are satisfied. (Also, (3) its behavior may change if the matches condition does not hold.)

Part II

1. Consider the following program changes where the previous version at the top is changed to the new version at the bottom according to the change contract in the middle.

[The previous version]

```
public class InterTypeMethodBinding extends MethodBinding {
  private MethodBinding syntheticMethod;

  public MethodBinding getAccessMethod() {
    return syntheticMethod;
  }
  /* the rest of the code is omitted */
}
```

[Change contract for getAccessMethod]

```
new_field postDispatchMethod:MethodBinding;
new_param staticReference:boolean;
matches staticReference == false;
```

[The new version]

```
public class InterTypeMethodBinding extends MethodBinding {
  private MethodBinding syntheticMethod;
  private MethodBinding postDispatchMethod;

  public MethodBinding getAccessMethod(boolean staticReference) {
    if (staticReference) return postDispatchMethod;
    else return [ syntheticMethod ];
  }
  /* the rest of the code is omitted */
}
```

Q1. Explain in English what the above change contract means:

(1) The new version getAccessMethod can access to a new field postDispatchMethod and a new parameter staticReference. (2) Its behavior should be preserved if the condition of the matches clause is satisfied. (Also, (3) its behavior may change if the matches condition does not hold.)

Q2. Also, fill in the blank of the new version.

2. Consider the following LazyMethodGen constructor.

```
1  public LazyMethodGen(Method m, LazyClassGen enclosingClass) {
2    this.enclosingClass = enclosingClass;
3    if (!m.isAbstract() && m.getCode() == null) {
4      throw new RuntimeException("bad non-abstract method with no code: " +
5        m + " on " + enclosingClass);
6    }
7    MethodGen gen = new MethodGen(m, enclosingClass.getName(),
8      enclosingClass.getConstantPoolGen());
9    this.memberView = new BcelMethod(enclosingClass.getType(), m);
10   this.accessFlags = gen.getAccessFlags();   this.returnType = gen.getReturnType();
11   this.name = gen.getName();   this.argumentTypes = gen.getArgumentTypes();
12   this.declaredExceptions = gen.getExceptions();   this.attributes = gen.getAttributes();
13   this.maxLocals = gen.getMaxLocals();
14   if (gen.isAbstract() || gen.isNative()) {
15       body = null;
16   } else {
17       body = gen.getInstructionList();   unpackHandlers(gen);
18       unpackLineNumbers(gen);   unpackLocals(gen);
19   }
20   assertGoodBody();
21 }
```

The above constructor creates a custom object representing a Java method. This constructor raises a RuntimeException (see line 4–5) if method m (i.e., the first formal parameter of the constructor) does not have its associated code for its body (see "m.getCode() == null" at line 3) when this method is expected to have a body. Otherwise, an object should be created successfully. Remember that a Java method does not have its body only when it is declared as either an abstract method or a native method. That is, the following method declarations are legal in Java programs. Notice that bodies are not provided for the methods.

```
public abstract void foo();
public native void bar();
```

The problem of the above LazyMethodGen constructor is that a RuntimeException is raised even when the given first parameter m represents a native method. Such behavior of the constructor is buggy because a native method does not have to have body code. Thus, instead of raising a RuntimeException, the constructor should create an object successfully. In other words, a RuntimeException should not be thrown.

Q. Based on the above description, write a change contract for the above constructor. You can use the following APIs if necessary.

- boolean isNative() of class Method, i.e., the class of the first formal parameter of the LazyMethodGen constructor:
  - This method determines whether the method is declared as native or not.

signaled (RuntimeException) m.isNative();
not_signals RuntimeException; OR signals (RuntimeException) false;

3. Consider the following program changes where the previous version at the top is changed to the new version at the bottom according to the change contract in the middle. Notice that the new version has an additional field droppingBackToFullBuild.

[The previous version]

```
public class AjPipeliningCompilerAdapter implements AbstractCompilerAdapter {
  List resultsPendingWeave = new ArrayList();
  private boolean reportedErrors;

  public void beforeCompiling(ICompilationUnit[] sourceUnits) {
    resultsPendingWeave = new ArrayList();
    reportedErrors = false;
  }

  /* the rest of the code is omitted */
}
```

[Change contract for beforeCompiling and the other methods]

```
new_field droppingBackToFullBuild: boolean;
matches droppingBackToFullBuild == [ false ];
```

[The new version]

```
public class AjPipeliningCompilerAdapter implements AbstractCompilerAdapter {
  List resultsPendingWeave = new ArrayList();
  private boolean reportedErrors;
  private boolean droppingBackToFullBuild; // a new field

  public void beforeCompiling(ICompilationUnit[] sourceUnits) {
    resultsPendingWeave = new ArrayList();
    reportedErrors = false;
    droppingBackToFullBuild = false; // a new statement
  }

  /* the rest of the code is omitted */
}
```

Depending on the boolean value of the new field droppingBackToFullBuild, the behaviors of the methods in AjPipeliningCompilerAdapter are either preserved or changed. Only if its value is true, the behaviors are changed. If its value is false, the new version behave in the same as the previous version does.

Q. Fill in the blank of the above change contract.